conference

proceedings

Conference on File and Storage Technologies

Monterey, California, USA
January 28–30, 2002

Sponsored by **The USENIX Association**
in cooperation with
ACM SIGOPS and **IEEE TCOS**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX

Proceedings of the Conference on File and Storage Technologies — *Monterey, CA, Jan. 28–30, 2002*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
USENIX URL: http://www.usenix.org
ALS URL: http://www.linuxshowcase.org

The price is $35 for members and $45 for nonmembers.
Outside the U.S.A. and Canada, please add
$14 per copy for postage (via air printed matter).

© 2002 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-03-0

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the FAST '02 Conference on File and Storage Technologies

January 28–30, 2002
Monterey, California, USA

Conference Organizers

Program Chair

Darrell Long, *University of California, Santa Cruz*

Program Committee

David Black, *EMC*
Walter Burkhard, *University of California, San Diego*
Randal Burns, *IBM Research & Johns Hopkins University*
Jeff Chase, *Duke University*
Richard Golding, *Panasas*
Roger Haskin, *IBM Research*
Peter Honeyman, *CITI, University of Michigan*
David Kotz, *Dartmouth College*
Mike Leis, *Quantum*
Ethan Miller, *University of California, Santa Cruz*
David Nagle, *Carnegie Mellon University*
M. Satyanarayanan, *Carnegie Mellon University*
Rodney Van Meter, *Nokia*
Randy Wang, *Princeton University*
John Wilkes, *Hewlett-Packard Labs*
Erez Zadok, *State University of New York at Stony Brook*

Steering Committee

David Anderson, *Seagate*
Garth Gibson, *Panasas & Carnegie Mellon University*
John Howard, *Sun Microsystems*
Merritt Jones, *Mitre*
Darrell Long, *University of California, Santa Cruz*
Kirk McKusick, *Consultant*
Jai Menon, *IBM Research*
David Patterson, *University of California, Berkeley*
Margo Seltzer, *Harvard University*

The USENIX Association Staff

External Reviewers

Alok Aggarwal
Guillermo Alvarez
Ahmed Amer
Darrell Anderson
Eric Anderson
Paul M. Aoki
Ismail Ari
Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau
Omer M. Asad
Ralph Becker-Szendy
Bill Bolosky
Rajesh Bordawekar
Peter Braam
Nick Burke
Steve Byan
Pei Cao
Rick Carlson
Ludmila Cherkasova
Tzi-cker Chiueh
Ariel Cohen
Toni Cortés
Preston Crow
Matt DeBergalis
Daniel Duchamp
Alexander Dupuy
Maria Ebling
Ted Faber
Mike Feeley
Jason Flinn
Liana Fong
William E. Freeman
Greg Ganger
Nitin Garg
Sumeet Garg
Timothy J. Gibson
Karen A. Glocer
Uday Gupta
John Hartman
John Heidemann
T. C. Hu
John Ioannidis
Naomaru Itoi
Mahesh Kallahalla
Christos Karamanolis
Kim Keeton
Terence P. Kelly
Jay Kistler
Olga Kornievskaia
Dejan Kostic
Arvind Krishnamurthy
Geoff Kuenning
Ed Lee
Hui Lei
Charles Lever
Gerry McAndrews
Patrick McDaniel
Patrick E. Mantey
Richard Martin
Jeanna Neefe Matthews
John May
David Mazières
Arif Merchant
Justin Moore
Susan Nagy
A. L. Narasimha Reddy
Dushyanth Narayanan
Jason Nieh
Nils Nieuwejaar
Ron Oldfield
Sean O'Malley
Hugo Patterson
David Pease
Fred Perner
Zachary Peterson
David Petrou
Niels Provos
Erik Riedel
Seth Robertson
Ben Rosin
Julian Satran
Wayne Sawdon
Steve Schlosser
Frank Schmuck
Thomas Schwarz
R. Sekar
Srini Seshan
Prashant Shenoy
Elizabeth Shriver
Sean Smith
Sumeet Sobti
Sara Sprenkle
Daniel Stodolsky
S. Sudarshan
Carl Tait
James Tanis
Bill Tetzlaff
Mustafa Uysal
Amin Vahdat
Alistair Veitch
Geoff Voelker
Kaladhar Voruganti
Richard Wheeler
Rajiv Wickremesinghe
Jim Wyllie
Bennet Yee
Tsozen Yeh
Ken Yocum
Haifeng Yu
Chi Zhang

FAST '02
Conference on File and Storage Technologies

January 28–30, 2002
Monterey, California, USA

Index of Authors v

Message from the Program Chair vii

Monday, January 28

Secure Storage
Session Chair: David Nagle, Carnegie Mellon University

Strong Security for Network-Attached Storage 1
Ethan L. Miller and Darrell D. E. Long, University of California, Santa Cruz; William E. Freeman, TRW; and Benjamin C. Reed, IBM Research

A Framework for Evaluating Storage System Security 15
Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan, Hewlett-Packard Laboratories

Enabling the Archival Storage of Signed Documents 31
Petros Maniatis and Mary Baker, Stanford University

Performance and Modeling
Session Chair: Randal Burns, IBM Research

WOLF—A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems 47
Jun Wang and Yiming Hu, University of Cincinnati

Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems 61
Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Timing-Accurate Storage Emulation 75
John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger, Carnegie Mellon University

Handling Disaster
Session Chair: Ethan Miller, University of California, Santa Cruz

Venti: A New Approach to Archival Storage 89
Sean Quinlan and Sean Dorward, Bell Labs, Lucent Technologies

Myriad: Cost-Effective Disaster Tolerance 103
Fay Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon E. Perl, and Li Zhang, Compaq Systems Research Center

SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery 117
Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara, Network Appliance Inc.

Tuesday, January 29

Wide-Area Storage
Session Chair: Rodney Van Meter, Nokia

Safety, Visibility, and Performance in a Wide-Area File System 131
Minkyong Kim, Landon P. Cox, and Brian D. Noble, University of Michigan

Obtaining High Performance for Storage Outsourcing 145
Wee Teck Ng, Bruce Hillyer, Elizabeth Shriver, Eran Gabber, and Banu Özden, Bell Laboratories

PersonalRAID: Mobile Storage for Distributed and Disconnected Computers 159
Sumeet Sobti, Nitin Garg, Chi Zhang, and Xiang Yu, Princeton University; Arvind Krishnamurthy, Yale University; and Randolph Y. Wang, Princeton University

Self-Organizing Storage Systems
Session Chair: Jeff Chase, Duke University

Hippodrome: Running Circles Around Storage Administration 175
Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch, Hewlett-Packard Laboratories

Selecting RAID Levels for Disk Arrays 189
Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez, and John Wilkes, Hewlett-Packard Laboratories

Appia: Automatic Storage Area Network Fabric Design 203
Julie Ward, Hewlett-Packard Laboratories; Michael O'Sullivan, Stanford University; and Troy Shahoumian and John Wilkes, Hewlett-Packard Laboratories

Aqueduct: Online Data Migration with Performance Guarantees 219
Chenyang Lu, University of Virginia; and Guillermo A. Alvarez and John Wilkes, Hewlett-Packard Laboratories

Wednesday, January 30

Parallel I/O
Session Chair: David Kotz, Dartmouth University

GPFS: A Shared-Disk File System for Large Computing Clusters 231
Frank Schmuck and Roger Haskin, IBM Almaden Research Center

Exploiting Inter-File Access Patterns Using Multi-Collective I/O 245
Gokhan Memik, University of California, Los Angeles; Mahmut Kandemir, Pennsylvania State University; and Alok Choudhary, Northwestern University

Low-Level Storage Optimization
Session Chair: Peter Honeyman, CITI, University of Michigan

Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics 259
Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger, Carnegie Mellon University

Freeblock Scheduling Outside of Disk Firmware 275
Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger, Carnegie Mellon University

Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload 289
Chi Zhang and Xiang Yu, Princeton University; Arvind Krishnamurthy, Yale University; and Randolph Y. Wang, Princeton University

Index of Authors

Alvarez, Guillermo A. 189, 219
Anderson, Eric 175, 189
Arpaci-Dusseau, Andrea C. 61
Arpaci-Dusseau, Remzi H. 61
Baker, Mary 31
Bucy, John S. 75
Chang, Fay 103
Choudhary, Alok 245
Cox, Landon P. 131
Dorward, Sean 89
Federwisch, Mike 117
Forney, Brian C. 61
Freeman, William E. 1
Gabber, Eran 145
Ganger, Gregory R. 75, 259, 275
Garg, Nitin 159
Griffin, John Linwood 259
Haskin, Roger 231
Hillyer, Bruce 145
Hitz, Dave 117
Hobbs, Michael 175
Hu, Yiming 47
Ji, Minwen 103
Kallahalla, Mahesh 15
Kandemir, Mahmut 245
Keeton, Kimberly 175
Kim, Minkyong 131
Kleiman, Steve 117
Krishnamurthy, Arvind 159, 289
Leung, Shun-Tak A. 103
Long, Darrell D. E. 1
Lu, Chenyang 219
Lumb, Christopher R. 259, 275
MacCormick, John 103
Maniatis, Petros 31
Manley, Stephen 117
Memik, Gokhan 245
Miller, Ethan L. 1
Ng, Wee Teck 145
Noble, Brian D. 131
O'Sullivan, Michael 203
Owara, Shane 117
Özden, Banu 145
Patterson, Hugo 117
Perl, Sharon E. 103
Quinlan, Sean 89
Reed, Benjamin C. 1
Riedel, Erik 15
Schindler, Jiri 75, 259, 275
Schlosser, Steven W. 75
Schmuck, Frank 231
Shahoumian, Troy 203
Shriver, Elizabeth 145
Sobti, Sumeet 159
Spence, Susan 175
Swaminathan, Ram 15, 189
Uysal, Mustafa 175
Veitch, Alistair 175, 189
Wang, Jun 47
Wang, Randolph Y. 159, 289
Ward, Julie 203
Wilkes, John 189, 203, 219
Yu, Xiang 159, 289
Zhang, Chi 159, 289
Zhang, Li 103

Message from the FAST Conference Chair

FAST '02 is an exciting new conference that fills the need for a high-quality research forum in the areas of file systems and storage systems. It takes the place of I/O in Parallel and Distributed Systems (IOPADS) and also provides a forum for research that might otherwise be scattered among many of the best operating system, distributed system, and computer architecture conferences.

The FAST Proceedings, 21 papers selected as the best of 110 submissions, represents outstanding work in the area by top researchers from both academia and industry. This 19% acceptance rate ranks it among the most selective conferences in computer science and reflects the excellent quality of the submissions. This extremely exciting conference presents the best of current research and provides a strong vision of the future. We also have several invited speakers, leaders from both academia and industry, laying out their visions of the future.

This conference started out as the result of the hard work and vision of a small cadre of researchers who saw the need for a high-quality conference. These early pioneers include Sean O'Malley, Peter Honeyman, Richard Golding, Randal Burns, and a few other supporters and confidants. It took several years, but at the 2000 OSDI conference we brought together our core group of people, about a dozen dedicated folks, and we came up with the plan to make the conference a reality. The list is too long to include here, but you know who you are, and please know that I am extremely grateful for your help. As a result of the hard work of this founding group, and subsequently the program and steering committees, I am confident that we have created the best file and storage system conference ever. I am sure you will enjoy the program, learn a lot, and be inspired.

Sincerely,
Darrell Long
Program Chair

Strong Security for Network-Attached Storage

Ethan L. Miller[†]
University of California, Santa Cruz
elm@cs.ucsc.edu

William E. Freeman
TRW
william.freeman@trw.com

Darrell D. E. Long[†]
University of California, Santa Cruz
darrell@cs.ucsc.edu

Benjamin C. Reed
IBM Research
breed@almaden.ibm.com

Abstract

We have developed a scheme to secure network-attached storage systems against many types of attacks. Our system uses strong cryptography to hide data from unauthorized users; someone gaining complete access to a disk cannot obtain any useful data from the system, and backups can be done without allowing the super-user access to cleartext. While insider denial-of-service attacks cannot be prevented (an insider can physically destroy the storage devices), our system detects attempts to forge data. The system was developed using a raw disk, and can be integrated into common file systems.

All of this security can be achieved with little penalty to performance. Our experiments show that, using a relatively inexpensive commodity CPU attached to a disk, our system can store and retrieve data with virtually no penalty for random disk requests and only a 15–20% performance loss over raw transfer rates for sequential disk requests. With such a minor performance penalty, there is no longer any reason not to include strong encryption and authentication in network file systems.

1 Introduction

Computer storage is an increasingly important part of the Internet, and ensuring the security and integrity of stored data is a crucial problem. Attacks by hackers and insiders have led to billions of dollars in lost revenue and expended effort to fix the resulting problems. Most organizations rely heavily on their distributed computing environment, which usually consists of workstations and a shared file system. This file system is typically stored on a centralized file server that is managed by a system administrator with super-user privileges, leaving the data vulnerable to anyone who can obtain (legitimately or otherwise) super-user access.

[†]Supported in part by Lawrence Livermore National Laboratory under contract B513238.

Recently, however, network-attached storage has begun to replace traditional centralized storage systems [1, 12]. In such systems, disks are attached directly to a network, and rely upon their own security rather than using the server's protection. This arrangement makes security more difficult because the disk is directly exposed to potential attacks instead of being hidden behind a single server that can be "hardened."

Most existing secure storage systems provide either authentication or encryption, but not both. For example, CFS [3] encrypts data, but does not easily permit authentication of data or sharing with other users. Systems such as SFS-RO [18] and NASD [12, 13] use encryption to provide network security and authentication, but store data in the clear. Recently, systems such as TCFS [6] and SUNDR [19] have incorporated both authentication and encryption, but at a relatively high penalty to performance.

We have developed a security system for network-attached storage that relies upon strong cryptography to protect data stored in a distributed storage system. Our system stores and transfers all data encrypted, only decrypting it at a client workstation. The drives lack sufficient information to decrypt the data they hold or to undetectably forge new data, so physically stealing the media will not enable an attacker to gain access to the data or to plant false data. Similarly, an administrator backing up the storage system has access to only encrypted copies of the data; the authorized users of a particular file are the only ones with access to its unencrypted contents.

Despite this level of security, our system does not impose much overhead on the file system. Our experiments using raw disks show that the encryption and verification provided by our system imposes almost no penalty for small random accesses to blocks on disk and less than a 20% penalty for large sequential transfers. Integration into a file system will further reduce this overhead by increasing the "base" time due to other file system overheads.

We begin by describing previous work in securing storage systems, discussing the strengths and weaknesses of each system. We then describe Secure Network-Attached Disks (SNAD), our system for protecting data on network-attached disks. Next, we describe the experiments we ran to test our systems performance and show that security for network-attached storage is possible without much performance penalty. We conclude with a description of our plans for integrating strong security into modern file systems.

2 Related Work

Many systems have been designed to address the security problems of modern distributed file systems. However, these systems have suffered either from weak security, poor performance, or both. It is only recently that CPU performance has advanced to the point where strong cryptography can be done quickly with inexpensive processors. This allows its use on low-cost processors that can be associated with each disk in a distributed file system [12].

2.1 Controlling Access to File Systems

Most file systems include some measure of security. However, systems such as xFS [1] and Petal [16] pass nearly all of their data in the clear, relying on relatively insecure networks and trusted hosts for data protection. Such a tactic works well if a network is totally disconnected from the rest of the world, but is a poor solution for modern systems that are exposed to the Internet. Some protection can be provided via firewalls or secure network protocols [4, 15], but these mechanisms do not protect data stored on disk. NFS offered little security until recently [22]; the new NFSv3 and NFSv4 protocols promise additional security, but there is little experience with the performance overheads of providing such security.

Other systems, such as AFS [14, 24] and NASD (Network Attached Secure Disk) [12, 13] use Kerberos [20] to provide security. These systems provide stronger security by requiring users to obtain "tickets" from a third party. The tickets are then presented to the file server (AFS) or NASD disk as proof of identity and access rights. These systems are considerably stronger than those that rely upon simple authentication, but they still suffer from several problems. First, files are left in the clear on the disks themselves, and may be transferred in the clear as well. Second, Kerberos-based systems rely upon a centralized security authority that is separate from the disks themselves. This is advantageous for sharing within a well-connected organization, but can become more difficult for widely distributed systems.

SCARED [21] is another file system that uses encryption to authenticate remote network storage. The SCARED design supports the use of end-to-end encryption of data, and, similar to SNAD, uses timestamps and counters to protect against replay attacks. However, SCARED does not implement end-to-end data encryption, leaving that for the underlying file system. SCARED, like the highest-performance version of our security system, uses secure hashes for authentication.

The Secure File System (SFS) [11, 18] provides strong authentication and a secure channel for communications. It also allows servers to authenticate their users and clients to authenticate servers. However, the general implementation of SFS [11] requires that users trust file systems to store and return file data correctly. SFS-RO [18] does not impose such a requirement, but it also forbids remote clients from writing to the file system, limiting writes to users on the server with access to the server's private key. The SUNDR file system [19] will address these issues by providing strong encryption and authentication for all file system users; however, its use of public-key encryption will subject it to the same performance issues we discuss in this paper.

2.2 Protecting Data on Disk

While most file system security has focused on access control and protecting data in transit, there have been a few file systems that have protected data on disk as well. There has been some work on protecting data on disk by making it impossible to delete [25]; however, our focus is on protecting data on disk from discovery by an intruder.

Many users have implemented their own "secure file system" by simply encrypting their files using standard encryption software. This provides confidentiality and, if the user also signs the file, a mechanism for ensuring that the server did not corrupt the data. However, this is an ad hoc mechanism, and does not deal with many issues such as sharing files between users.

The Cryptographic File System (CFS) developed at AT&T Bell Laboratories [3, 5] encrypted all data and potentially sensitive metadata stored on disk. When a user desired access to an encrypted directory, he issued a command to attach the encrypted directory to a subdirectory of `/crypt`. If the correct password was entered, the data was subsequently available in decrypted form. Because the structures to support this were stored in a "normal" directory structure, they could be used with NFS and other file systems. However, CFS also required that the server be trusted to "actually store (and eventually return) the bits that were originally sent to it." In the Internet era, there is no guarantee that a server will do this, so there must be a mechanism to ensure that the server

has not maliciously altered the data. In addition, CFS does not discuss mechanisms for distributing keys among users for sharing files. A more recent cryptographic file system, Cryptfs [27] works in a similar way and has similar sharing and authentication issues.

Recently, TCFS [6] has provided strong security and authentication for file system users. However, TCFS is relatively slow, reducing file system performance by more than 50%.

The design of a trusted database system such as Trusted DataBase (TDB) [17] could be adapted to file systems; however, TDB is not easily scalable, making it less useful for large-scale file systems.

3 System Design

The goal of our system is to address the security shortcomings of previous file systems while preserving the flexibility and performance of standard distributed file systems. We propose three security alternatives for network-attached storage; the first two are considerably more CPU-intensive because they make extensive use of public-key encryption, but are also more secure. The third alternative avoids the use of public-key encryption on each block transfer, resulting in high performance on current low-cost CPUs while providing nearly as much security as the first two alternatives.

3.1 Design Goals

Our security schemes provide several important features for a secure file system. The first feature is end-to-end encryption of all file system data, including storage on disk. This is necessary to restrict access to data to only authorized users, specifically excluding system administrators and backup systems. An adversary with full access to all of the bits on the disk or the network should be unable to decipher any user files—the disk must not contain sufficient information to decrypt the data stored on it. Rather, data should only exist in unencrypted form on the client.

A second desirable feature is data integrity. A user reading data from the server must be sure that the files received are those originally stored. It is no longer a good idea to trust that a disk is secure against intruders; data modified at the disk or introduced into the system by an malicious intruder must be detectable. Storing a non-linear checksum over the cleartext in a block along with the ciphertext, as described in Section 3.4.3, allows any authorized user to detect a change made to the encrypted block by an intruder who did not have the symmetric key to encrypt the file.

Flexibility is a third feature that is desirable in a secure file system. While it would certainly be possible to simply encrypt each file with a user's password, this approach is impractical because it makes file sharing difficult. Instead, a file system should have sharing at least as powerful as that in standard UNIX and preferably as flexible as the access control lists provided by AFS [14].

High performance and scalability is the fourth feature desirable for a secure distributed file system. Though it may be possible to build a secure file system, users may avoid using it if performance is poor. If encryption and decryption are performed at the client, encryption throughput will limit a single client's bandwidth, but not the bandwidth of the entire system. By minimizing the effort required by the network-attached disk's CPU, however, it is possible to build a distributed storage system that can be used by hundreds of clients, each of which can decrypt the data intended for itself.

3.2 Basic Mechanisms

The basic mechanism behind our security system is to encrypt all data at the client and give the server sufficient information to authenticate the writer and the reader sufficient information to verify the end-to-end integrity of the data.

SNAD relies upon several standard cryptographic tools. The client uses the RC5 algorithm [23] to encrypt the data before it leaves the client, though any strong and fast algorithm such as Rijndael [7] would also be acceptable. This ensures that the data is unreadable by anyone until it is decrypted by the client that reads it. Public-key cryptography is used to allow disks to store information that can be used to decrypt their files; because public-key encryption is asymmetric, however, only a user with the appropriate private key can use this information. This process is described in Section 3.4. The security provided by SNAD is very strong; the symmetric algorithms use 128 bit keys—the key length Schneier recommends for highly secure information with a lifetime longer than 40 years [23]. If 128 bit keys are too short, longer keys may be used.

SNAD also makes extensive use of cryptographic hashes and keyed hashes. Cryptographic hashes such as MD4, MD5, and SHA-1 [23] use a one-way function to compute a large number (128 or 160 bits) from a block of data. Any modification in the input data will cause the resulting hash value to change. While it is possible to find two sets of input data that will result in the same MD4 hash (weak collision) [8], there is still no known way to produce a second input that hashes to the same value as a given first input. MD5 and SHA are variations on MD4 for which it is currently believed NP-hard to find two input texts that result in the same hash value.

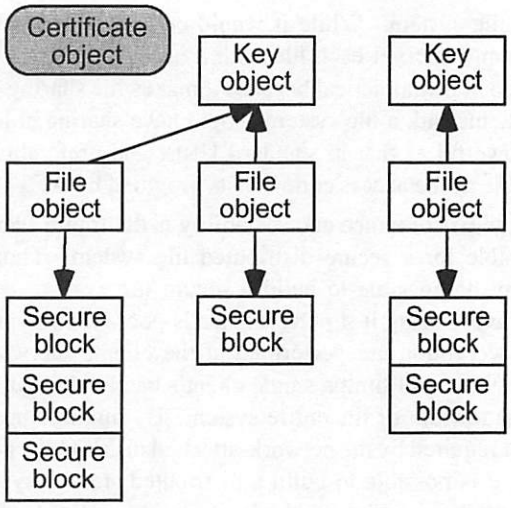


Figure 1: Relationships between objects in a Secure Network-Attached Disk.

Keyed hashes such as HMAC [2] use a cryptographic hash in conjunction with a shared secret to check integrity and authenticate a writer. If the sender and receiver share a key, the key can be included in the cryptographic hash, preventing anyone who intercepts the data from undetectably modifying it unless they know the shared key.

3.3 SNAD Data Structures

All of the SNAD security schemes use four basic structures: secure blocks, file objects, key objects, and certificate objects. Although these objects are all shown as contiguous blocks of data, there is no requirement that they be stored contiguously on disk.

3.3.1 Overall Data Structure Organization

The overall data structure organization of SNAD is shown in Figure 1. The diagram shows multiple file objects using a single key object; this corresponds to a situation where two files have the same access controls. It is likely that there will be relatively few key objects on a disk, just as there are relatively few unique groups in a standard UNIX file system.

All of the objects shown in Figure 1 require relatively little overhead. Each data object requires 36–100 bytes of overhead, depending on which security scheme is being used. Even for 100 bytes of overhead, using 4 KB blocks requires just 2.4% overhead for cryptographic metadata. File objects require little overhead just a pointer to a key object. Key objects are also small: a key object requires 76 bytes for the header and 72 bytes for each user. If each of 10,000 users is part of 200 different groups, there will need to be 148 MB of key objects, or 0.37% of a 40 GB disk. The certificate object requires less than 300 bytes per user, adding just 3 MB to the total. Thus, all of the security information for SNAD occupies less than 3% overhead for a 40 GB disk. For comparison, the inodes in a UNIX file system typically consume 1–2% of total storage.

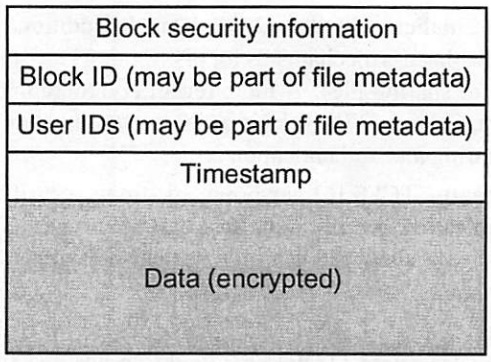


Figure 2: Secure block.

3.3.2 Secure Blocks

A secure block (SB) is the minimum unit of data that can be read or written in the secure file system, and corresponds to a file block in a standard file system. Files are composed of one or more secure blocks; a sample secure block is shown in Figure 2.

The block security information is different for each of the three security schemes discussed in Section 3.4, but is on the order of 32 bytes long. The block ID is a unique identifier for the block in the file system, and is a combination of the unique file identifier and block number in the file. The user ID is the creator of the secure block and is used by the SNAD server to determine which public key or writer authentication key to use to check the security of the block. If the server is an object-based storage device or file server, the user ID list need not be stored for each secure block; instead, it can be retrieved from the file or object to which the secure block belongs.

The data stored in the data object is encrypted using a symmetric encryption algorithm such as RC5. The key used to encrypt the data is obtained from the key object associated with the file, as described in Section 3.3.4. An initialization vector (IV) consisting of the file ID and block offset within the file is used to prevent identical plaintext blocks encrypted with the same key from encrypting to the same ciphertext. Knowledge of the IV does not aid in cryptanalysis of the block's ciphertext; rather, it prevents an attacker who cannot decrypt a secure block from determining which secure blocks contain the same plaintext.

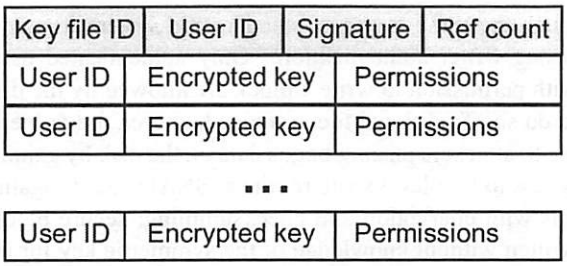


Figure 3: Key object.

The timestamp is used simply to prevent replay attacks; it need not be an actual timer, but instead could simply be a counter incremented at each client.

If a secure block is too large, each file will waste relatively large amounts of space on average, half of the last secure block. However, minimizing both storage and operational encryption overheads requires that objects not be too small. Like file blocks, secure blocks could be variably sized within a single file system; however, we assumed fixed sized secure blocks. We explore the performance tradeoffs with respect to object size in Section 4.

3.3.3 File Objects

File objects are composed of one or more secure blocks along with per-file metadata. In addition to the usual file metadata such as block pointers, file size, and timestamps, a file object contains a pointer to a key object. This pointer is used to find the keys that may be used to access the file. Except for the pointer to the key object and perhaps pointers to the extra information for secure blocks, the structures for file objects are identical to those for standard files.

3.3.4 Key Objects

Each key object, shown in Figure 3, contains several types of information. The key file ID is just the unique identifier for the key object on the system. The user ID in the header of the key object is that of the last user to modify the key object. The reference count is kept by the system to know when the key object is no longer needed.

When a user writes the object, he hashes the entire object except for the reference count and signs the hash with his private key, storing the result in the signature field. Anyone using the key object verifies the integrity of the object by performing the same hash and verifying the provided signature. This mechanism prevents the disk, or anyone with access to it, from undetectably modifying the security fields of a key object a client using the key object can check to ensure that the signature on a

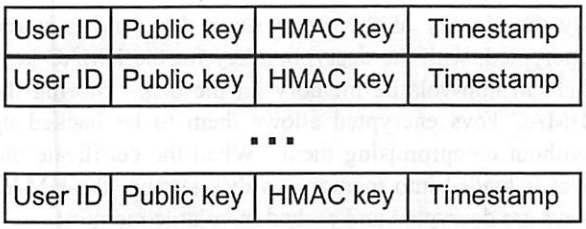


Figure 4: Certificate object.

key object belongs to someone authorized to change the key object. Because someone who modifies a key object must sign it, there is a way of tracing illegitimate modifications to a particular user.

Each tuple in the body of the key object includes a user ID, encrypted key, and permissions for that user. The user ID need not correspond to a single user; it could, instead, be an equivalent to a UNIX group and correspond to several users with shared access to a single private key, similar to the mechanism in TCFS [6]. The second field in the tuple contains the key for the symmetric RC5 algorithm. Rather than storing this key in the clear, the key object stores the key encrypted with the user's public key. The disk cannot decrypt any key unless it obtains a user's private key, but the only way to get a user's private key is to steal it from a client or the user himself because keys are kept on the client and never sent to the disk. The permissions field is used by the disk to determine whether the user is allowed to write the key object.

A key object may be used for more than one file. If this is done, all files that use the key object are encrypted with the same symmetric encryption key and are accessible by the same set of users. In this way, a key object corresponds to a UNIX group.

3.3.5 Certificate Objects

Each network-attached disk contains a single certificate object, shown in Figure 4, which contains administrative and cryptographic information about each SNAD user. The disk uses the information in the certificate object to authenticate users and do basic storage management.

The certificate object contains a list of tuples, each of which includes a user ID, public key, HMAC key (for Schemes 2 and 3), and timestamp. The user ID identifies the user or group to which the remainder of the tuple pertains. The public key is stored on the disk for two reasons: as a convenience so that the disk and those using it need not consult a centralized key server, and for writer authentication in one of the security schemes described in Section 3.4.

The HMAC key is used in two of the schemes to ver-

ify the identity of the user writing data, and is stored encrypted, with the decryption key for the HMAC keys held in non-volatile memory on the disk. Storing the HMAC keys encrypted allows them to be backed up without compromising them. When the certificate object is loaded into memory on disk startup, the HMAC keys are decrypted and cached in volatile memory.

The timestamp field is updated each time a user writes a file object, and is used to prevent replay attacks. A centralized clock is not necessary unless requests for a particular user ID may come from several clients at about the same time. This can occur if a user ID actually corresponds to a group, or if a user is logged on to several systems at once. The sole purpose of the timestamp is to prevent replay attacks; clocks may be synchronized using any number of common approaches, or replay attacks may be thwarted as described in Schneier [23]. An attacker who obtained a decrypted copy of the certificate object would be able to write to any block of the disk as if he had physical access to the disk. Attacks of this sort could destroy valid data by overwriting it, but could not plant undetectable fakes unless the attacker were also an authorized reader of the file (and even this is impossible if a block must be signed by its writer, as we require in two of our security schemes).

3.4 SNAD Security Schemes

Our security schemes all use symmetric encryption to encrypt data objects, but vary in the mechanisms used to provide end-to-end data integrity. This variation trades off slight reductions in integrity guarantees for significantly higher performance by varying the number, type, and location of the cryptographic operations. We focus on the operations performed in each of the schemes; details on the security of the schemes can be found in an earlier paper [10].

All of the SNAD protection schemes provide strong security by encrypting each block of data using RC5 at the client; other encryption algorithms may also be used. Because the RC5 keys are stored on the drive encrypted with the public key of any user permitted to access the file, even gaining access to both the ciphertext on the disk and the encrypted keys would be of no use without the necessary private key. As a result, the disks provide an encrypted block of data and encrypted keys to anyone who requests them. Assuming that the encryption is sufficiently strong, the encrypted information will not benefit an attacker, so there is little use in having the disk attempt to verify the identity of a requester. If the user can decrypt the symmetric key, he can obtain the block's plaintext.

Writing blocks in all three schemes is controlled in much the same way as a standard file system, but with strong writer authentication. Only authenticated users with permission to write a block are allowed by the disk to do so. Traditional file systems, however, are vulnerable to attackers placing bogus data on the disk by gaining access to low-level write routines. SNAD guards against this with encryption and checksumming; secure blocks written without knowledge of the symmetric key for the object will give a checksum error when decrypted by a client. The only way for an unauthorized write to occur is for an authorized reader to gain physical access to the disk, use the file's symmetric key to write a secure block, and (for Schemes 1 and 2) sign the cryptographic hash. This weakness is present in any security scheme that uses symmetric key encryption to protect files: anyone that can decrypt the file can encrypt it as well. Reading and writing data in each of the three schemes have much in common. First, the user must give his private key to the client, which is assumed to be trusted by the user. This can be done via password, authentication server (*e.g.*, as is used in Kerberos [20]), or smartcard. For each file, the user opens the file and reads the key object for the file; for this operation as any others, file system caching may be transparently used. The appropriate field of the key object is then decrypted the to obtain the symmetric encryption key for the file. This key is then used to encrypt the data before sending it to the server and after decrypt it after receiving it from the server.

3.4.1 SNAD Scheme 1

The first SNAD scheme provides security on each block of data similar to that provided by some cryptographic electronic mail security schemes such as PGP [28]. Writes in this scheme encrypt each data block, compute a hash over the entire data object (including the metadata), and sign the hash using the user's private key. This hash can then be verified by anyone with the user's public key. In particular, the disk can recompute the hash and compare it against the hash signed by the user who sent the block. If they match, the disk successfully verifies the provided signature, and the user has the permission to write the file, the SNAD server writes the block to disk. The block security information for this scheme thus consists of a signed secure hash.

Reads in this scheme require no operations by the SNAD server CPU, but do require that the client CPU check the hash and signature just as the SNAD server did on a write. Additionally, the client must decrypt the data.

Table 1 summarizes the operations that must be done for each read and write request. Note that this scheme requires relatively expensive signature and verification operations for each disk request; in particular, the CPU

Operations	Read		Write	
	Client	NAS	Client	NAS
En/Decrypt	×		×	
Hash	×		×	×
Signature			×	
Verification	×			×

Table 1: Cryptographic operations used in Scheme 1.

Operations	Read		Write	
	Client	NAS	Client	NAS
En/Decrypt	×		×	
Hash	×		×	×
Signature			×	
Verification	×			

Table 2: Cryptographic operations used in Scheme 2.

on the network-attached disk must perform an expensive signature verification for each block write. Because this CPU is likely to be slow, the verification will reduce write performance

3.4.2 SNAD Scheme 2

Scheme 2 replaces the SNAD server's signature verification with an HMAC. In this scheme, the client performs a cryptographic hash on the block and signs it. However, this signed hash, which is stored with the secure block, is only verified by the client when it reads the block. The client also calculates an HMAC on the secure block using the secret HMAC key it shares with the server and sends the HMAC to the SNAD server. The SNAD server computes an HMAC using the shared secret key from the certificate object and checks it against the HMAC received from the client. Recalculating the entire hash including the HMAC key would be time-consuming; instead, the client simply performs an HMAC over the hash.

The replacement of a signature verification by an HMAC reduces the load on the SNAD disk CPU, but does not reduce the load on the client CPU, which still must perform signatures on writes and verifications on reads. Table 2 shows the operations that the client and server perform for secure block reads and writes

3.4.3 SNAD Scheme 3

The previous two schemes use a public-key signature to identify the originator of a data block and ensure that the block hash has not been modified. The third scheme uses a keyed-hash (HMAC) approach to authenticate a

Operations	Read		Write	
	Client	NAS	Client	NAS
En/Decrypt	×		×	
Hash	×	×	×	×
Signature				
Verification				

Table 3: Cryptographic operations used in Scheme 3.

writer of a data block and verify the block's integrity. HMACs differ from signed hashes in that a user able to verify a keyed-hash is also able to create it. Scheme 3 still uses public-key authentication for key objects because writing key objects, while slower with public-key controls, is very infrequent.

Write operations in this scheme require the client to encrypt the secure block and calculate an HMAC over the ciphertext. This information is then sent to the disk, which authenticates the sender by recomputing the HMAC using the shared secret key from the certificate object. If the write is authentic and the user has the permissions to modify or create the secure block, the SNAD disk commits the write to disk, updating structures as necessary. Note that the disk does not store the HMAC because it must recalculate a new HMAC if the reader is a different user from the user who wrote the block.

Unlike the previous two schemes, this scheme requires the SNAD disk to perform a cryptographic operation on a read: the disk must calculate a new HMAC using the key from the user requesting the data. The data object, along with the new HMAC, is then sent to the client requesting the data. If the disk were forced to write blocks without the proper encryption key, a client could detect this during a read by recomputing the non-linear checksum over the cleartext and comparing it to the stored checksum.

The operations performed by the client and SNAD disk are summarized in Table 3. Note that this scheme requires no signature generation or verification operations; however, the SNAD disk must now compute an HMAC on both reads and writes

3.5 SNAD Design Issues

There are many design issues that must be considered when building a secure file system, particularly in the area of key management. Mazières, *et al.* discuss many of these issues in more detail [18, 19]; however, we feel that there are a few problems of particular importance that should be mentioned here. These issues include creating key objects, adding and removing users from a key object, and providing a key escrow system.

3.5.1 Creating a Key Object

The creation of file objects and data objects is relatively straightforward, assuming that an appropriate key object and certificate object already exist. However, there must be a way to create new key objects.

The primary requirement for a new key object is a new RC5 key that will be used to encrypt files that use the key object. The key object creator must ensure that the RC5 key is truly random (not merely pseudo-random), and then encrypt it with his own public key as well as that of anyone else he wishes to have access to the file. Once this is done, the key object may be stored on a SNAD disk, and is ready for use. This procedure is relatively simple, and only relies on the ability to generate truly random numbers for the RC5 key.

3.5.2 Modifying Access Permissions

One of the largest difficulties with many systems for maintaining security is dealing with the modification of access groups. Adding users to an access group is relatively straightforward a user with the rights to add a new user can simply use his private key to obtain the RC5 key, and encrypt that key with the new user's public key. The new user can now access the files associated with this key object.

Revoking permissions is a more difficult problem for which there are several possible solutions. The first solution is to simply delete the user's line from the key object; if this is done, the user will be unable to obtain a new copy of the RC5 key, though he may still have the RC5 key cached somewhere. A second solution is to immediately reencrypt the associated files using a different key object containing only those users who should still have access to the file. This solution is slower, but will ensure that the revoked user cannot access the file. A third solution is to apply the second solution lazily. This allows the revoked user to continue to access old data until the files are reencrypted, but denies him access to any new data, which is encrypted with a different key.

The choice of revocation method is still an open issue with no well-accepted solution. We are currently investigating tradeoffs between these three mechanisms for changing access permissions.

3.5.3 Key Escrow

One potential problem with an encrypted file system is that a user may abscond with his key (or simply lose it), making it impossible to access files that only he was allowed to see. In many organizations, this is an important argument against encryption.

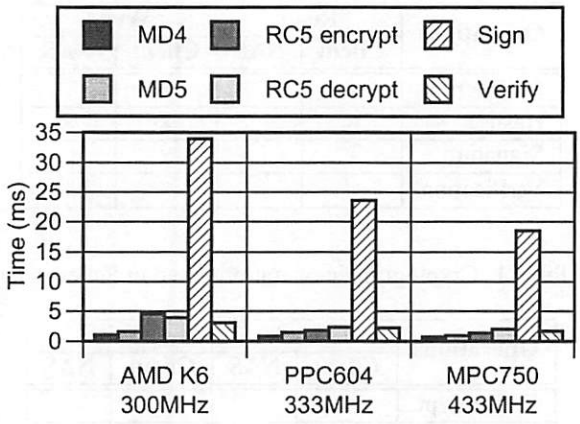


Figure 5: Performance of cryptographic algorithms on low-cost CPUs. Block size is 32 KB except for sign & verify, which are done on 128 bit inputs.

However, this problem can be solved with key escrow: including an escrow "user" in every key object. This private key for this escrow "user" may be kept in a safe (or even spread across multiple safes); the system only requires that the corresponding public key be available for the creation of entries in new key objects. This solution in no way weakens the strong security present in the file system; an intruder would still need the private key (which is not kept online) to break into any file.

Note that escrow is *not* required in SNAD, though it may be included if desired.

4 Performance

All of the security schemes we presented would go a long way towards securing data in distributed file systems. However, few would use such strong security if doing so meant crippling the file system's performance. Our measurements show that strong security can be achieved without sacrificing performance. Using slightly longer keys has relatively little effect on encryption speed, but doubles the time required for brute-force cryptanalysis for each bit added to the key length.

4.1 Cryptographic Overhead

We first tested the raw speed of the cryptographic algorithms used by SNAD; this provided insight into how fast each of our schemes was likely to be. We previously found that using encryption in time-critical systems is feasible [9]; performance tests on additional (newer) hardware are summarized in Figure 5.

As Figure 5 shows, the most expensive operation by far is signature generation. We used a modulus of 512 bits in the RSA algorithm, with 32,767 as the public



Figure 6: Cryptographic overhead for SNAD using a 360 MHz MPC750 for both client and disk, assuming 32 KB data blocks.

exponent, which allowed verification times to be much faster than signature generation times. Similar tests on a 200 MHz Pentium Pro with 1024 bit keys [26] required 43 ms for a public key signature; the faster processors available today should be able to complete this operation in times similar to those we measured for 512 bit keys.

The length of time required to compute a signature suggests that Schemes 1 and 2 are likely to be considerably slower than Scheme 3 on a workload that includes many writes. On a read-mostly file system, however, the long time required to calculate a signature is less important and the benefits of the stronger protection available from Schemes 1 and 2 may be more important. While this data was measured on relatively modern CPUs, progress marches on. As a result, a 500 MHz AMD K6 is currently available for $20 retail; a 300 MHz K6 is even less expensive, and both are inexpensive enough to serve as an embedded processor.

By combining Tables 1, 2, and 3 and Figure 5, we can derive the theoretical overhead for each security scheme. Figure 6 shows the overhead for each scheme if the MPC750 (PowerPC G3) is used in both client and server; different processors will have different overheads, but the ratios between the schemes will be similar.

From Figure 5, we can derive the theoretical "speed limit" for performance using a 360 MHz MPC750 (PowerPC G3) for both client and disk. Schemes 1 and 2 are limited to nearly 6.4 MB/s for reads, but only 1.4 MB/s for writes. Scheme 3, on the other hand, can read at up to 10 MB/s and write even faster—12.7 MB/s. These rates are based on cryptographic overhead only; they do not include network and disk delays. However, they are useful in showing how fast a cryptographic file system could go given sufficiently fast disks and networks. Note, too, that Schemes 1 and 2 are limited primarily by the amount of time needed by the client to compute the signature; thus, they may work well in environments with many clients and relatively few disks.

4.2 SNAD Performance Measurements

Though measuring the performance of cryptographic operations is useful, it does not show the full impact of end-to-end security on a distributed file system. We constructed prototype SNAD disks and clients, and ran experiments to see how much performance degradation was incurred when cryptographic overhead was added to a block-level SNAD server. The observations in this section present the worst-case scenario for cryptographic overheads because real file systems will likely have other overheads not present in a raw block server, allowing the cryptographic overheads to be partially overlapped with file system overheads.

Our workload consisted of reads and writes to logical blocks on disk with two access patterns: random and sequential. For the random access pattern, the client accessed a randomly selected a sequence of secure blocks. In the sequential access pattern, the client made 4 MB sequential requests, broken up into individual requests for secure blocks. This access pattern minimized seek and rotational latency but still incurred cryptographic overhead for each secure block.

Our experimental setup consisted of multiple VME boards running a real-time kernel (Wind River's VxWorks^{TM}). Each board was based on the MPC750 running at either 333 or 360 MHz. The VME chassis was used only for power; the boards were connected to each other by 100 Mbit/s Ethernet switched through a Cisco 2900XL switch. In addition, each server was connected to a Seagate Cheetah 10K RPM UltraSCSI disk drive. We used 360 MHz boards for both client and server for the one-to-one tests; our multiple client and server tests used different configurations that are detailed later.

4.2.1 Baseline: No Security

Our first set of tests stressed the system without any cryptography, showing how fast the system could read and write data unencrypted and unencumbered by any security mechanisms. Figure 7 shows the performance of a one client, one disk SNAD system without any cryptographic overhead. There is a knee in the performance curve around 8 KB, and a block size of 32 KB delivers nearly the maximum performance permitted by a 100 Mbit/s Ethernet for sequential access. As expected, random accesses are slower than sequential accesses, though the large write buffer on the disk allows write performance for random writes to approach that of sequential writes for large blocks.

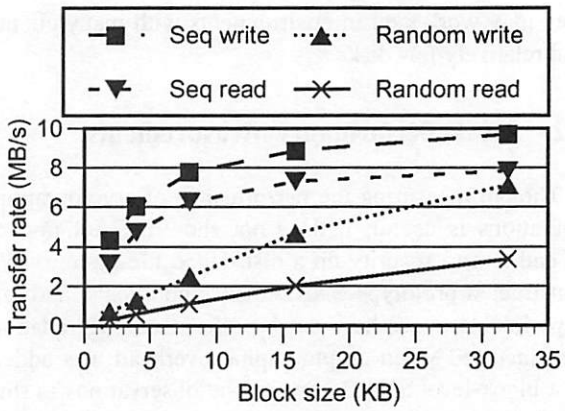


Figure 7: SNAD performance without cryptographic controls.

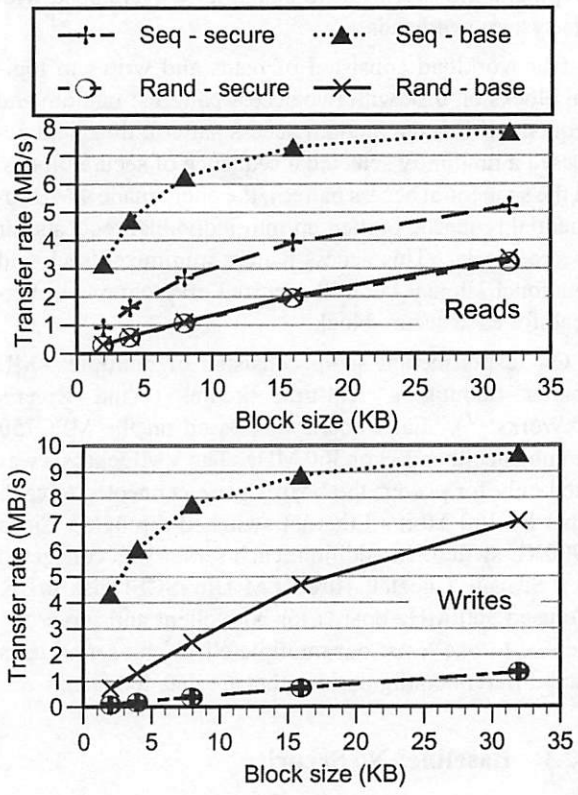


Figure 8: SNAD performance using Scheme 1.

We used the performance measurements shown in Figure 7 as a baseline for our other performance measurements, showing the effect of strong cryptographic security on file system performance for each security scheme in Section 3.4.

4.2.2 Performance of Scheme 1

As described in Section 3.4.1, Scheme 1 provides the best security, albeit at the cost of lower performance. Our

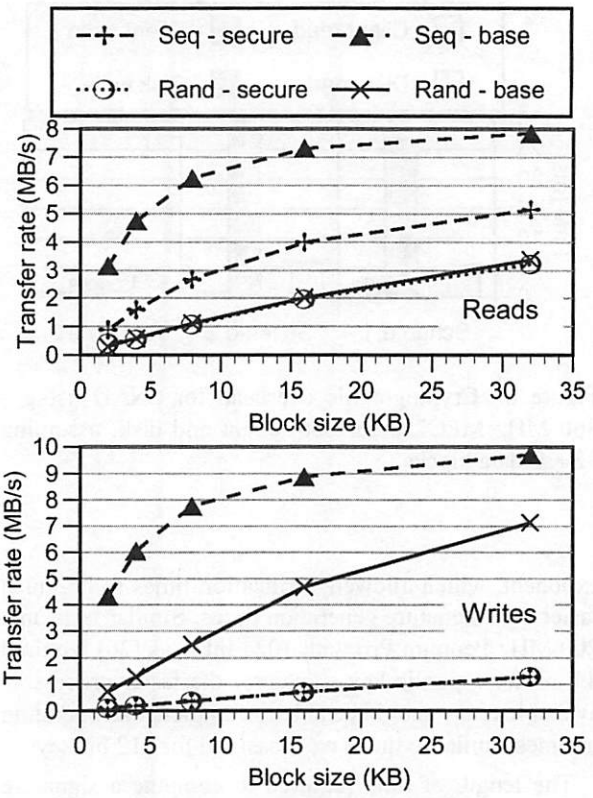


Figure 9: SNAD performance using Scheme 2.

experiments showed that, as expected, Scheme 1 suffers greatly on both sequential and random writes. However, Scheme 1 can keep up with random reads of blocks up to 32 KB, though it cannot keep up with sequential reads. These results are shown in Figure 8.

The performance shown in Figure 8 indicates that, with current processors, Scheme 1 is unsuitable for distributed file systems that require good performance with one exception: file systems that are dominated by small random (non-sequential) reads. For most access patterns, though, we must use other security schemes until processor speeds increase sufficiently to permit use of Scheme 1.

4.2.3 Performance of Scheme 2

Scheme 2 improves upon the first scheme by changing the write operation to be less CPU-intensive at the SNAD server with little loss in security. The read operations in both Schemes 1 and 2 are identical, and the graph in Figure 9 indeed shows that the two schemes perform identically, with sequential reads suffering a significant performance loss and random reads running at the same speed encrypted and in the clear. However, the hoped-for performance gains on writes did not materialize with a single client because the bottleneck was in the gener-

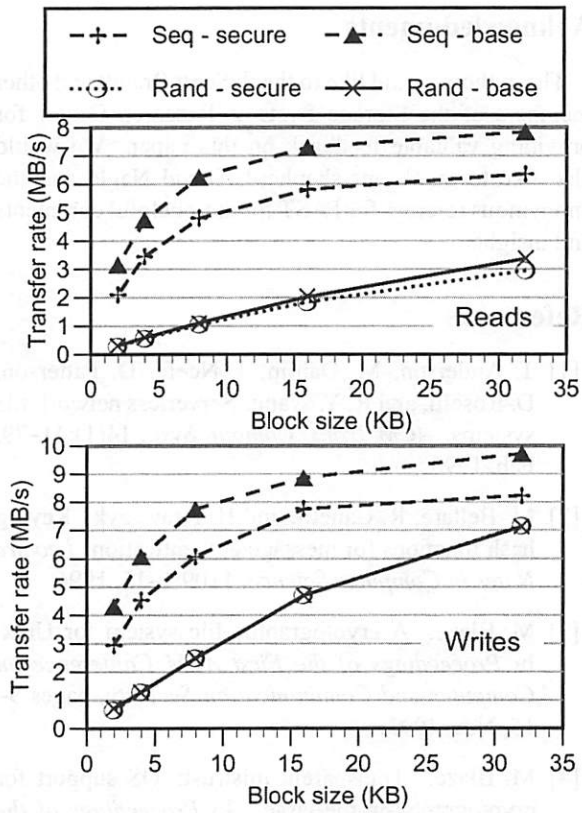


Figure 10: SNAD performance using Scheme 3.

ation of the public-key signature at the client. Instead, the write performance of Scheme 2 is similar to that of Scheme 1; neither is currently suitable for systems with large sequential writes.

4.2.4 Performance of Scheme 3

Scheme 3 replaces the signed hash for block integrity and writer authentication with a keyed hash (HMAC). While this results in slightly less security, performance for this scheme is greatly improved over the first two schemes, as shown in Figure 10. This graph shows that, for Scheme 3, random I/O operations (read and write) suffer little or no performance penalty for cryptographic controls with block sizes between 2 KB and 32 KB. Long sequential transfers, on the other hand, do suffer a small performance penalty: large sequential writes with encryption run at 88% of the bandwidth of unencrypted writes, and large sequential reads run at 81% of the bandwidth of unprotected reads. We believe that this relatively small performance penalty is an acceptable price to pay for a large increase in file system security.

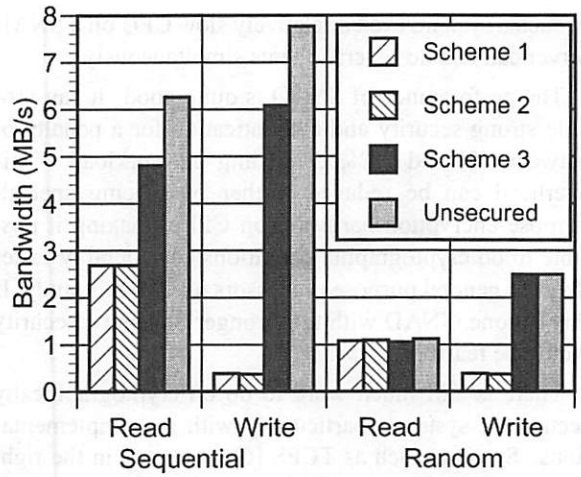


Figure 11: Performance of three security schemes and unsecured operations for 8 KB blocks.

4.2.5 Performance Summary

Figure 11 shows the performance of all three security schemes and unsecured storage in a system using 8 KB blocks. We chose this block size because, although many current UNIX systems use 4 KB blocks, we believe that 8 KB (or even larger) is an appropriate size in an environment where 40 GB disks are common. In a system dominated by small reads, any of the security schemes would be acceptable, and would not reduce performance significantly.

In systems with many sequential operations or even a moderate number of writes, however, only the third scheme maintains performance within 20% of unsecured storage. The first two security schemes require the client to generate a public-key signature on writes, limiting performance. Sequential reads under the first two schemes also have reduced performance due to the public-key signature verification required on the client. This operation is much faster than signing, and does not slow down random reads, though it is not fast enough for sequential reads.

5 Future Work

We are currently building a large-scale file system using object-based storage devices that use the security system described in this paper. Using this testbed, we are investigating the scalability of the different security schemes. Schemes 1 and 2 are slow in part because the clients must generate a signature. With one client and one server, this reduces performance. However, with many relatively low-bandwidth clients, the overhead of generating signatures is distributed to many machines.

In such a system, even a relatively slow CPU on a SNAD server can handle several clients simultaneously.

The performance of SNAD is quite good: it can provide strong security and authentication for a penalty of between 1% and 20%, depending on workload. This overhead can be reduced further by placing special-purpose encryption hardware on CPUs, making it possible to do cryptographic operations considerably faster than the general purpose processors used in this study. If this is done, SNAD with the stronger Scheme 1 security would be feasible.

There is still much work to do on cryptographically secure file systems, particularly with real implementations. Systems such as TCFS [6] are a step in the right direction; however, issues such as performance, key revocation and security infrastructure in general need to be explored further.

6 Conclusions

We presented a design for Secure Network Attached Disks and demonstrated that strong security for storage need not drastically reduce system performance. Random access reads and writes in our system suffered almost no performance penalty, and large sequential operations ran at 88% of maximum for writes and 81% of maximum for reads. This performance was achieved using inexpensive CPUs which could be included on each secure disk.

This security mechanism for distributed storage systems solves many of the performance and security problems in existing systems today. This system provides user data confidentiality and integrity from the moment it leaves the client computer. The distributed storage system should perform substantially better than centralized file servers, and provide better reliability. Having the security functionality decentralized will improve performance and scalability and remove the single point of failure that plagues many proposed centralized security schemes to date.

Integrating SNAD security schemes into modern distributed file systems is essential. Unsecured data is vulnerable to threats ranging from security holes in the operating system to unscrupulous users with access to raw storage devices. Implementing the security schemes we have described in a storage system costs relatively little in performance while providing tremendous advantages in security. Given the hostile environment on the Internet, distributed storage systems can no longer afford to be without strong security.

Acknowledgments

The authors would like to thank Scott Brandt and other members of the Storage Systems Research Center for providing valuable feedback on this paper. We would also like to thank our shepherd, David Nagle and the anonymous referees for FAST for their helpful comments and insights.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, Feb. 1996.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Lecture Notes in Computer Science*, 1109:1–15, 1996.

[3] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the First ACM Conference on Computer and Communication Security*, pages 9–15, Nov. 1993.

[4] M. Blaze. Transparent mistrust: OS support for cryptography-in-the-large. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 98–102, 1993.

[5] M. Blaze. Key management in an encrypting file system. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 27–34, June 1994.

[6] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In *Proceedings of the Freenix Track: 2001 USENIX Annual Technical Conference*, pages 199–212, Boston, MA, June 2001.

[7] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Proceedings of the Third Smart Card Research and Advanced Applications Conference*, 1998.

[8] H. Dobbertin. Cryptanalysis of MD4. *Lecture Notes in Computer Science*, 1039:53–69, 1996. Fast Software Encryption Workshop.

[9] W. Freeman and E. Miller. An experimental analysis of cryptographic overhead in performance-critical systems. In *Proceedings of the 7th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '99)*, pages 348–357, College Park, MD, Oct. 1999. IEEE.

[10] W. Freeman and E. Miller. Design for a decentralized security system for network-attached storage. In *Proceedings of the 17th IEEE Symposium on Mass Storage Systems and Technologies*, pages 361–373, College Park, MD, Mar. 2000.

[11] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–196, San Diego, CA, Oct. 2000.

[12] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Interational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[13] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, July 1999. Also available as Technical Report CMU-CS-99-160.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. Wes. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[15] J. Ioannidis and M. Blaze. The architecture and implementation of network-layer security under Unix. In *Proceedings of the First ACM Conference on Computer and Communication Security*, pages 29–39, Nov. 1993.

[16] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th Interational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, 1996.

[17] U. Maheshwari, R. Vingralek, and B. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150, San Diego, CA, Oct. 2000.

[18] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Dec. 1999.

[19] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 99–104, Schloss Elmau, Germany, May 2001.

[20] B. C. Neumann, J. G. Steiner, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 191–201, Dallas, TX, 1988.

[21] B. Reed, E. Chron, R. Burns, and D. D. E. Long. Authenticating network attached storage. *IEEE Micro*, 20(1):49–57, Jan. 2000.

[22] J. Reid. Plugging the holes on host-based authentication. In *Computers and Security*, pages 661–671, 1996.

[23] B. Schneier. *Applied Cryptography*. Wiley, New York, NY, 2nd edition, 1996.

[24] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Trans. Comput. Syst.*, 14(2):200–222, May 1996.

[25] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, Oct. 2000.

[26] M. J. Wiener. Performance comparison of public-key cryptosystems. *RSA CryptoBytes*, 4(1), Summer 1998.

[27] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, 1998.

[28] P. Zimmerman. *PGP Source Code and Internals*. MIT Press, 1995.

A framework for evaluating storage system security

Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan
Hewlett-Packard Laboratories
Palo Alto, California
{riedel,maheshk,swaram}@hpl.hp.com

Abstract

There are a variety of ways to ensure the security of data and the integrity of data transfer, depending on the set of anticipated attacks, the level of security desired by data owners, and the level of inconvenience users are willing to tolerate. Current storage systems secure data either by encrypting data on the wire, or by encrypting data on the disk. These systems seem very different, and currently there are no common parameters for comparing them. In this paper we propose a framework in which both types of systems can be evaluated along the security and performance axes. In particular, we show that all of the existing systems merely make different trade-offs along a single continuum and among a set of related security primitives. We use a trace from a time-sharing UNIX server used by a medium-sized workgroup to quantify the costs associated with each of these secure storage systems. We show that encrypt-on-disk systems offer both increased security and improved performance over encrypt-on-wire in the traced environment.

1 Introduction

Much of the focus of recent storage security work has been on protecting communication between clients and servers in an untrusted, networked world [Gobioff98, Kent98, Mazieres99, Satran01]. In particular, the focus is on protecting data *integrity*: preventing unauthorized modification of commands or data, modification of requests in transit, and replaying of requests. Some of these systems further address the issue of *privacy*, or confidentiality, of data transfer: preventing the leaking of data in transit by snooping on the network.

The most comprehensive treatment of this topic is Network-Attached Secure Disks (NASD) [Gobioff99a], which uses capabilities provided to users by a file manager separate from the storage servers. A barrier to wide acceptance of the NASD scheme is the performance cost of the encryption and integrity checking needed at both clients and servers. In order to reduce this cost, NASD proposes a scheme using *pre-computed checksums* with secure hashes [Gobioff99] that pre-calculates and stores checksums for long-lived data. NASD does not provide a comparable scheme to optimize privacy since this would require *pre-computed encryption*. If data were stored on the server in encrypted form, then it would not be necessary to encrypt it for each transfer on the network. The difficulty with such a scheme is that encryption in NASD is done using session keys generated for each client/server interaction, whereas pre-computation requires longer-lived keys.

From the client's point of view, these two schemes are identical – it receives encrypted data and must pay the cost of checksumming and decrypting it. From the point of view of an adversary, they are also equivalent – the data he sees is encrypted and unintelligible. The difference is only whether the server has to bear the encryption cost each time a new session key is chosen, or whether it can take advantage of data already stored in encrypted form. Similarly, if written data is encrypted before it leaves the client and is stored encrypted, the server eliminates any decryption work.

Storing data in encrypted form was originally proposed in Blaze's Cryptographic File System (CFS) and expanded in later systems [Blaze93, Cattaneo97, Zadok98, Hughes99], where it is used for a different purpose – to protect data from untrusted servers. If data is stored on the server in encrypted form it is protected from leaking by the server (who does not know the key), and there is no need to encrypt data again when it is sent on the network. Encryption is done by the original creator of the file, and updated by subsequent writers, but the server performs no encryption or decryption. Secure checksums are still needed to ensure the integrity of the communication, but privacy is ensured without repeated per-byte encryption[1]. In order to use the data, users must still decrypt it, but using a long-term key that must now be obtained *a priori*.

To support sharing in a system that encrypts data on disk, the problem is simply one of key distribution – how users obtain these long-term keys. This can be done via a centralized server such as the NASD file manager or an NIS server. Alternatively, a distributed scheme where data owners provide keys to eventual users directly, as would have to be done for a system such as CFS, removes a cen-

[1] If desired, privacy of arguments still requires encryption of message headers, but not of bulk file data.

tral point for attack. A variant of such a key distribution scheme is proposed in SFS [Mazieres99, Fu00] and further expanded in the Cepheus file system [Fu99]. The SNAD system [Miller02] combines aspects of both CFS (on-disk encryption) and SFS (secure communication and authentication) into a single encrypt-on-disk system.

Even though many secure storage systems have been proposed and described individually, there is no systematic way to compare and contrast them. We remedy this situation by presenting an agnostic framework to describe the features of these systems and the level of security they offer. Any secure storage system must implement a core set of functions, although they may vary in the detailed design choices. These choices affect both the level of security that the system provides, and the performance the system achieves. A similar study has been done to establish a framework for evaluating digital certificate revocation mechanisms [Iliadis00].

In addition to security and performance, there is a third factor to consider when building any secure system: the level of inconvenience users are willing to tolerate. If users must type in a separate password for every document they open, or individually choose access rights for every file they create, they will soon begin to circumvent the best intentions of the system designers [Whitten99]. Precise metrics to gauge the impact of this effect are not yet established, so we will treat this issue only indirectly.

Given our framework, we show how to quantitatively compare the performance of previously proposed systems, the overhead on users, and the security guarantees that the systems offer. We do this using a trace from a non-secure UNIX file system to estimate the work required for the various secure schemes. This evaluation is independent of the actual system implementations, and provides a general way of evaluating security and estimating cost. Finally, our analysis shows that encrypt-on-disk systems are not only more secure but also provide better performance than encrypt-on-wire systems.

The rest of the paper is organized as follows. Section 2 defines our framework for storage system security, identifies a range of attacks, and suggests a core set of security primitives. Section 3 describes how system designs proposed elsewhere fit into the framework, and how the choices they make impact security or improve performance. Section 4 evaluates the decisions made along each of these axes using a traced workload from a UNIX time-sharing server to concretely quantify security costs in day-to-day usage. Finally we conclude in Section 5.

2 Framework of storage security

In this section, we abstract the commonalities among known secure storage systems into a general framework. The framework consists of five components: players, attacks, security primitives, granularity of protection, and user inconvenience. We elaborate each of these next.

2.1 Players

Here we define the players we use in the rest of the paper. This list covers all of the possible players that one has to consider for protecting stored data. Each player is listed with a set of legitimate actions it can perform. Any other action by that player is treated as an attack.

a) ***owners*** – create and destroy data (i.e., render data un-readable by all readers), delegate read and write permission to other players, and revoke another user's privilege to read or write owned data.

b) ***readers*** – read data once permission to read was delegated by owners.

c) ***writers*** – modify data once permission to write was delegated by owners.

d) ***wire*** – transfers data between other players.

e) ***storage servers*** – store and return data upon request. (For instance, these are file servers in NFS, disks in NASD, or disk arrays in iSCSI.)

f) ***group servers*** – authenticate other players and authorize access based on membership groups as defined by *owners*. (For instance, these are group servers in NASD or the NIS server in NFS.)

g) ***namespace servers*** – allow traversal of namespaces, such as provide support for lookup of directories and files in directories.

Finally, we define an *adversary* to be any entity who attempts to perform functions other than those that it is authorized to. Notice that this definition of adversary also includes legitimate players attempting to perform actions beyond what they are authorized to.

Though in the above definitions, functionalities differentiate players, actual systems might choose to aggregate multiple players into a single entity. For instance, NASD combines the functionality of the group server and namespace server into a single metadata server.

We intentionally omitted any key-escrow agent from the list of players because its main purpose is to reveal keys and identities when necessary but it does not add to the basic level of security of a storage system.

At this point, it is important to note that the framework presented here is not intended to allow evaluation of the end-to-end security of a particular system. This requires careful analysis of each system component and the particular combination of components. Any secure system is only as strong as its weakest link. Our framework is no replacement for such analysis, but simply seeks to allow a high-level comparison among different schemes, purposely leaving some secondary details unexamined.

2.2 Attacks

Broadly, there are two kinds of data that players handle:

- short lived data that is communicated, or agreed upon, in each session, and
- long lived data and metadata for persistent storage.

Existing systems for network security have mostly addressed the compromise of short-lived data and the protocols used to communicate them. In addition to securing data on the wire, storage systems must also secure long-lived data on the servers. These two requirements give rise to the following set of attacks. The attacks may be mounted on the data or the metadata, unless explicitly specified otherwise:

a) ***by the adversary on the wire*** – for instance, an attack mounted on the NASD protocol used to communicate files to the clients.

b) ***by the adversary on the servers*** – for instance, an adversary updating a file on a NFS file server.

c) ***by a revoked user on the servers*** – for instance where a revoked reader (no longer part of the system) can continue to read files in Cepheus.

d) ***by the adversary colluding with the storage server*** – for instance, one where a CFS encrypted directory is deleted by the UNIX file system.

e) ***by the adversary colluding with the group server*** – for instance an adversary gaining access to data after corrupting a NASD file manager.

f) ***by the adversary colluding with readers or writers*** – for instance, a reader passing a copy of a file to an adversary.

The last attack, involving collusion with other readers or writers is very difficult to prevent without substantial complexity and support from outside the system, and has been listed above for completeness. We will not consider it further in this paper.

Each of the above attacks can further be broken into three kinds based on the effect they have on the data:

a) ***leak attacks*** – are those where the adversary gains access to some data.

b) ***change attacks*** – are those where the adversary makes valid modifications to data (i.e., modifications that readers cannot detect as invalid).

c) ***destroy attacks*** – are those where the adversary makes invalid modifications to some stored data. An invalid modification is any change to data that is detectable as incorrect by the owner or readers.

Table 1 provides a summary of these attacks and where they occur in practice. The data summarizes a survey of CIOs and system managers showing the percentage of respondents reporting a particular attack. The table shows the primary types of attacks from our list above that each of these real-world attacks touches. The intent is to motivate the importance of all of the attacks listed above, including some that may not have been considered very crucial in past work (such as revocation).

2.3 Core security primitives

Secure storage systems as proposed in research and commercial systems implement a myriad of security features to enable players to securely perform their functions. Though the details of the schemes used differ, the core

	frequency	*cost*	*messages*		*data*				
attack	*% of companies reporting*	*estimated damage ($ millions)*	*leak*	*change*	*leak*	*change*	*destroy*	*revoked user*	*denial of service*
telecom eavesdropping	10 %	1	X	–	–	–	–	–	–
active wiretap	2 %	n/m	–	X	–	–	–	–	–
system penetration	40 %	19	X	X	X	X	X	–	–
laptop theft	64 %	9	–	–	X	–	X	–	–
theft of proprietary information	26 %	150	–	–	X	–	–	X	–
unauthorized access by insiders	49 %	6	–	–	X	X	–	X	–
sabotage	18 %	5	–	–	–	–	X	–	X
virus	94 %	45	–	–	–	–	X	–	–
denial of service	36 %	4	–	–	–	–	–	–	X

Table 1. *Frequency and cost of attacks.* The frequency of various attacks and their mapping into our framework. The % numbers are as reported in a survey of five hundred system managers taken in Spring 2001, with almost all categories showing significant increases over previous years [Power01]. The cost column gives the self-estimated damage to their businesses. Note that although over 75% of respondents claimed that they had experienced some monetary damage due to the attacks reported, only 35% were able to estimate the extent of the damage, which means the numbers shown are only low estimates. Industry estimates of the total damage to companies worldwide from all attacks run into the billions of dollars. The boxes marked "X" show the primary damage caused by a particular attack, although other damage is also possible in many cases. The intent is not to exhaustively enumerate the damage, but to motivate each of the attacks in the framework as an important threat and give a very rough idea of relative importance.

set of security primitives can be abstracted into six types: authentication, authorization, securing data on the wire, securing data on the disk, key distribution, and revocation. As we show in Section 3, not all systems necessarily provide support for all of these, and the choices made directly affect the performance of the system and security guarantees provided. In the rest of this section we elaborate on each of these primitives and the important choices in implementing them.

2.3.1 Authentication

The purpose of authentication is to establish the identity of a particular player in order to authorize their actions. Storage systems may implement authentication in one of two general ways:

a) ***distributed authentication*** – owners explicitly authenticate each player to authorize access to the data they own (as in CFS, or the use of server public keys in SFS).

b) ***centralized authentication*** – owners delegate responsibility for authentication and authorization to a group server (as accomplished through checks done by the file server in NFS or the file manager in NASD).

In general, there are three mechanisms to achieve mutual authentication: a public key infrastructure (PKI), a centralized scheme (e.g., Kerberos [Steiner88]), or a password-based scheme. The former two are quite similar. Both need a trusted third party and differ in how often this party is consulted. The latter one requires some pre-exchanged shared secret, which can be difficult to maintain in a distributed environment.

The usual concern is about authentication of owners, readers, and writers to storage servers or group servers, but there may also be concern about authenticating *servers* to *users* to prevent improper service. Again, although this is an important consideration, we do not consider it a primary security requirement for this analysis

2.3.2 Authorization

The purpose of authorization is to allow the owner of some data to delegate (partial) access to the data to another player. The user is authenticated and the identity checked against a known set of permissions determined by data owners. Authorization can be done in one of two general ways:

a) ***server-mediated*** – servers receive actions and perform them on behalf of readers, writers, and owners (as in NFS and AFS).

b) ***owner-handled*** – owners provide readers and writers with keys that they can use to authorize or perform actions (such as the capabilities in NASD, and the server keys in SFS).

2.3.3 Securing data on the wire

Protocols for ensuring reliable and secure passing of messages have been well studied. Several standard protocols have been proposed, including SSL to protect web traffic, SSH to protect remote terminals, and IPsec to protect Internet traffic more generally [Kent98]. A variant of such a system for storage is used in NASD [Gobioff98]; a similar scheme is used in the self-certifying file system [Mazieres00]; and IPsec has been proposed as the security mechanism for iSCSI [Satran01].

To ensure data integrity on the wire some scheme involving keyed checksums (MACs) will always be needed, irrespective of the design chosen. The MAC is used to tie the checksum to a particular player, and the checksum is used to tie the MAC to a particular set of data. A timestamped MAC also protects against replay or server impersonation (man-in-the-middle) attacks [Gobioff98].

With the increasing deployment of protocols such as SSL and IPsec, hardware solutions are becoming available that offload the heavyweight cryptographic operations from client or server processors. Such hardware may support an entire protocol in its end-to-end form, or simply provide accelerated primitives that can be used in different ways by various systems. Once concerns over raw encryption or checksum speed are removed, parameters such as number of key changes and requirements for key storage present further bottlenecks [Cravotta01].

2.3.4 Securing data on disk

The reasons one may want to encrypt data on the disk are that the server is inherently untrusted or the server might be compromised, such as a stolen disk or laptop. To guarantee that the data and metadata are not compromised, they must be stored encrypted on disk. To accomplish this encryption, two types of ciphers may be used:

a) ***symmetric cipher*** – a single private-key system, such as DES or AES [Schneier95, Nechvatal00], that is used to perform bulk data encryption and decryption (such as the privacy option in NASD).

b) ***asymmetric cipher*** – a system using a pair of keys, such as RSA [Schneier95], that is generally used for authentication and to bootstrap the shared keys to be used by the symmetric cipher (such as the authentication protocols of IPsec).

Since computing asymmetric ciphers is much slower than symmetric ciphers, these operations are used sparingly, either for key exchange, or to protect stored symmetric keys in a *lockbox* (such as those used in Cepheus).

2.3.5 Key distribution

In a secure storage system that relies on encryption to protect data, each piece of data has some associated keys – either symmetric of asymmetric, depending on the

structure of the system – that are required to access it. These keys may be used in one of two ways to:

a) ***directly encrypt data*** – the keys are used by writers to encrypt and by readers to decrypt data directly at the edges of the system (as in CFS).

b) ***prove authorization*** – possession of the keys is used by readers and writers to prove that they have the requisite authorization (such as the capabilities in NASD).

Use of the keys to prove authorization requires the server be trusted to accurately perform the necessary checks. Direct encryption ensures that only readers or writers are able to access the data or create valid new data. However, it complicates revocation since readers and writers have been given the keys themselves, rather than simply delegated capabilities.

2.3.6 Key distribution

For either use of keys, any system with shared access to files then requires some mechanism to distribute keys among readers and writers. Current systems implement this key distribution in one of two ways:

a) ***using a group-server*** – a centralized group server maintains the keys to all files, and the access control lists. If a user is in a particular list, then the server provides the key to the corresponding file (as in NFS, AFS, and Cepheus).

b) ***owner-handled*** – file owners themselves provide readers and writers with keys that they can use to perform actions. This typically complicates key revocation if the readers and writers cache keys (as in variants of CFS).

2.3.7 Revocation

Traditionally revocation is discussed in the context of centralized services such as certificate authorities (CAs) where it removes the association between the physical identity of a player and a particular key. In the context of secure storage, this is extended so that a player's access privileges to a particular piece of data can be revoked. When a player is revoked (e.g., a user leaves a particular workgroup) the keys to which this player had access must be changed. In systems where data is stored encrypted, this will require data to be re-encrypted, which may be done as follows:

a) ***aggressive re-encryption*** – immediately after a revocation, re-write data with a new key. Copies of data distributed under the old key in the past remain readable.

b) ***lazy re-encryption*** – delay re-encryption of the file to the next time it is updated [Fu99] or read. This saves encryption work for rarely-accessed files, but leaves data vulnerable longer.

c) ***periodic re-encryption*** – change keys and re-write data periodically to limit the window of vulnerability [Gobioff99a].

The distinction between aggressive and lazy re-encryption is a general consideration for secure storage. If a user had access to particular data at one time, they may anyway have copied it elsewhere, so protecting future changes becomes most important.

2.4 Granularity of protection

To provide secure storage, a system bears the additional overhead of the cryptographic operations discussed above, and the key management. To limit the key overhead, various systems implement different optimizations including aggregation of players into groups to simplify authorization, and trading off the security of short-lived keys against the ease of management of long-term keys.

2.4.1 Group membership

The purpose of group membership is to compactly represent the permissions on a particular set of data by simply verifying the membership of a player in a group, and then authorizing access based on group permissions. There are two ways to decide group membership, namely:

a) ***distributed group membership*** – owners explicitly determine who is authorized to share data and distribute the necessary keys (as in CFS).

b) ***centralized group membership*** – owners delegate authorization to a group server that distributes keys (as in NFS with NIS, NASD, and Cepheus).

Access control lists are a variant of group membership that explicitly enumerate all the players, but these ACLs must still be stored somewhere and essentially provide the group membership function [Howard88, Hughes99].

2.4.2 Granularity of keys

The keys used to encrypt and decrypt a particular set of data may be short-term or long-term. Short-term keys reduce the vulnerability window by decreasing the amount of data encrypted with the same key, whereas long-term keys are easier to manage since there are fewer of them, and they are exchanged less often.

a) ***short-term keys*** – typically last for the duration of one player and one session (as in NASD, and iSCSI with IPsec).

b) ***long-lived keys*** – typically last across sessions and might be the same across multiple players (as in CFS and SFS).

When using long-term keys, the granularity of data associated with a single key greatly impacts the number of keys required; the choices include a key per-file, per-directory, per-user-group, or per-file-group.

Additional concerns arise when considering very long-lived keys, such as digital signatures on documents that must last for many years [Maniatis02], or for backup tapes [Boneh96].

2.5 User inconvenience

In addition to security and performance, it is critical to consider the level of inconvenience users are willing to tolerate before they become sloppy and circumvent the intent of the system:

a) ***convenient*** – single login password and tokens derived from this.

b) ***inconvenient*** – compartmentalized access, multiple passwords for different services, passwords are re-entered frequently and changed regularly.

c) ***very inconvenient*** – resources are protected at a very low level (e.g., password per document opened or per application invocation).

Forcing users to remember long lists of passwords often leads to poor password choices, or sloppy password practices (e.g., post-it notes) [Adams99]. The problem is exacerbated when users handle keys explicitly and make encryption choices on their own [Whitten99].

Some of the password issues may be addressed by widespread use of smart cards. The difficulty is that this removes the main aspect of active user involvement in maintaining security. Users must be aware of security in some way, otherwise they will become complacent and assume the system is infallible. The parameters of these trade-offs are not yet well understood, but overall security of data may well hinge on such usability issues [Whitten99].

3 Secure storage systems

In this section we cast previously proposed designs for secure storage onto the framework described in Section 2. Where appropriate, we highlight the trust assumptions made by each design, and mention specific extensions proposed. Our intent is to evaluate each system against a common set of criteria. For this reason, we concentrate on those aspects that address the primary functions of a secure storage system. This does not mean that additional functions or characteristics of individual designs are less important. The overall security of a system must always be evaluated holistically: a system is only as secure as its weakest link.

In this same vein, we also assume that issues of operating system trust are dealt with separately. For all the discussion in this paper, readers, writers and owners should be thought of as the smallest possible *trusted core* surrounding a user [Dalton01]. If necessary, this may even be a smart card or other protected device that handles all encryption and key storage.

The comparison in Table 2 summarizes the characteristics of each of the systems presented in this section, and which attacks each system addresses.

3.1 CFS and similar systems

The first widely-known discussion of security for storage systems is the Cryptographic File System (CFS) [Blaze93]. In CFS a directory to be protected is encrypted using a secret key. The underlying data is then stored as a single file in the host file system and attached as a cleartext directory under a */crypto* mount point. This allows the host file system to treat the encrypted data as yet another file. Normal utilities such as backup function without alteration; they never have access to the cleartext data. The system is implemented as a user-level NFS loopback mount, and files are decrypted when accessed.

CFS was designed as a secure *local* file system, so it lacks features for sharing encrypted files among users. The only way to share a protected file is to directly hand out keys for protected directories to other users. However, CFS does protect against attacks where the bits on disk are compromised, such as when a computer is stolen. The key characteristics of CFS are:

players

- owners, readers and writers are indistinguishable.
- the host file system acts as the storage server as well as the group server, in authorizing file access.
- namespace traversal is handled by readers and writers themselves.

trust assumptions

- the storage server is untrusted and does not access the keys, protecting against leak and modify attacks involving collusion with the storage server.
- the storage server is trusted to prevent destruction of data.

security primitives

- owners handle authentication when distributing keys to encrypted directories and files.
- authorization for read is done by passing keys to readers and writers.
- the host file system verifies the authorization of writers to overwrite existing data, but the validity of these modifications is assured only by having the proper key.
- writers encrypt data using a symmetric cipher before storing it on disk.
- there is no provision to protect data while on the wire, as CFS is essentially a local system.

- since CFS is designed for the local file system, distribution of keys is done directly by the owners.
- revocation requires immediate re-encryption of data, since a revoked user can collude with the storage server to attack the data.

granularity

- the local file system aggregates users into groups to authorize access, but there is no explicit decision on aggregating the keys used to encrypt data.
- long-lived keys are used on a per-directory basis.

CryptFS [Zadok98] extends CFS to be more efficient by building it as a stackable file system rather than a user level server. It attempts to make the system more resilient to attacks due to corruption of individual users by using session IDs and user IDs to index into the key table, rather than using only usernames. TCFS [Cattaneo97, Cattaneo01] uses a lockbox to store a single key (rather than per-directory keys), and encrypts only file data and file names; directory structures and other metadata are left un-encrypted. Beyond the implementation differences and varying key granularity, CryptFS, TCFS, and CFS are identical with respect to our framework. All of these systems are described for use on a local file system. They could also be used as mounts over a remote file system, with protection of the communication to the remote server. We consider only the simple, local case here.

A later generation CFS [Blaze94] includes a key escrow system. This is necessary to recover keys when they cannot be obtained from the owner, for instance, after an owner has left the organization. Truffles [Reiher93] uses an alternative method of handling this problem by splitting keys such that any n members of a group can collude to regenerate the key of a missing owner.

All of the above systems assume untrusted servers; keys are known only to the owners, readers and writers, and not trusted to the system itself. The key escrow system in CFS depends on trusting the key database, but not trusting the servers. Truffles distributes this trust so that a group of owners are trusted instead of a single database.

There are several systems that encrypt data on entire devices and transparently decrypt the data when it is accessed. These include Secure Drive [Swank95], Secure FileSystem [Gutmann96] and PGPdisk [NA98]. These systems are similar to CFS except that they themselves do not perform any authentication or authorization; they rely on the operating system for these primitives.

3.2 SFS

Most secure storage systems assume the servers to be part of the trusted infrastructure and concentrate on guaranteeing that the users accessing the servers are properly authenticated. The Secure File System (SFS) [Mazieres99] addresses the problem of *mutually* authen-

			adversary alone			with storage server			by revoked user		other	
system	S#	message attacks	leak	change	destroy	leak	change	destroy	leak	change	subvert group server	denial of service
CFS & similar	3.1	–	yes	yes	no	yes	yes	no	–	–	–	no
SFS	3.2	yes	yes	yes	yes	no	no	no	no	no	no	no
SFS-RO	3.3	yes	yes	yes	no	yes	yes	no	no	–	yes[2]	no
Cepheus	3.4	yes	yes	yes	yes	yes	yes	no	yes[1]	yes	no	no
SNAD	3.4	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no
NASD	3.5	yes	yes	yes	yes	no	no	no	yes	yes	no	no
iSCSI w/ IPsec	3.6	yes	yes	no	no	no	no	no	yes	yes	–	no
LUN security	3.7	no	no	no	no	no	no	no	no	no	–	no
AFS	3.8	yes	yes	yes	yes	no	no	no	yes	yes	yes	no
NFSv4	3.9	yes	yes	yes	yes	no	no	no	yes	yes	no	no
Windows EFS	3.10	–	yes	yes	no	yes	yes	no	–	–	no	no
PASIS/S4	3.11	–	–	–	yes	yes	yes	yes	–	–	–	no[4]
OceanStore	3.11	–	yes	yes	yes	yes	no[3]	yes	yes	yes	–	no[4]

Table 2. *Summary of security guarantees provided by different systems.* A "yes" means that the system prevents that particular attack; for instance, Cepheus prevents attacks that leak data by stealing the storage server because it encrypts on the media. A "no" means that the system fails to handle that particular attack. A dash means the attack is not applicable to that system. (1) Cepheus uses lazy revocation, which re-encrypts data only on the next update; this allows data to leak until is has been updated, making this a qualified "yes". (2) Subverting the group server does not open any additional vulnerabilities that are not already present from the adversary acting alone. (3) Since only a single replica is used by each reader, a reader colluding with a single storage server could cause another reader to see invalid modifications. (4) Although a request to a busy replica could be re-directed to other replicas, a combined attack on all the replicas could still be mounted.

ticating the servers and users. Authentication of the server is necessary to prevent an adversary spoofing the server, for instance, when the servers are part of a public infrastructure. One important tenet of SFS is that it is independent of the key distribution and authentication mechanisms. The characteristics of SFS are:

players

- owners, readers and writers are differentiated.
- the storage server also functions as the group server in authenticating users.

trust assumptions

- the storage server is trusted with the data and is vulnerable to leak or modify attacks by an adversary colluding with the server.

security primitives

- servers and users perform mutual authentication. Servers are authenticated using *self-certifying* pathnames to files. Self-certifying pathnames are similar to mount points in traditional NFS, except that they have the public key of the server embedded in them.
- the group server uses NFS style user authorization.
- a session key is used to protect all communication between the server and users.
- a distributed mechanism is used to obtain server keys (through self-certifying pathnames).
- revocation of servers requires readers to check a centralized *revocation list* of revoked servers.

granularity

- traditional UNIX style aggregation of users into groups helps simplify authorization.
- uses a session key to protect all communication

3.3 SFS-RO

SFS was extended in SFS-RO [Fu00] to support storage and retrieval of encrypted read-only data. This provides a solution to securely distribute widely-accessed data (such as application binary kits) over the Internet using individually insecure mirrors as storage servers. SFS-RO has the following characteristics:

players

- same as SFS except that there are no writers – only owners can modify the data that they have created.

trust assumptions

- the storage server is not trusted with the data and hence not vulnerable to leak or modify by the adversary in collusion with the server.

security primitives

- same as in SFS, except that data is stored encrypted on the disk. Data is signed and encrypted by the owners when it is stored. Readers can verify the integrity of data by verifying the signature.

granularity

- since the data is already encrypted on disk, there is no need to encrypt it again before transmission.

3.4 Cepheus and SNAD

The Cepheus system [Fu99] builds on SFS to develop a general purpose file system, while Secure Network-Attached Disks (SNAD) [Miller02] combines the functions of CFS and SFS. In particular, both systems keep files encrypted on disk, and include the ability to share and update the encrypted data. They differ only in a few areas, and have the following characteristics:

players

- owners, readers and writers are differentiated via specific authorization schemes for writes.
- Cepheus uses separate storage servers and a group server that distributes lockboxes. SNAD relies on public/private key pairs for groups and must use a group server to distribute these, but stores lockboxes directly on storage servers.

trust assumptions

- the storage server is not trusted with the data and hence not vulnerable to leak or modify attacks by an adversary in collusion with the server.
- the storage server holds file encryption keys in lockboxes that are encrypted. In Cepheus, only readers and writers hold the keys to lockboxes, preventing attacks in collusion with the group server. In SNAD, separate key pairs are used for groups, so the group server for these is vulnerable.
- revoked users can continue to decrypt files until the files are updated, at which point they are encrypted with a new key (*lazy revocation*). Revoked users cannot update or destroy data.

security primitives

- servers check user authentication and authorization via the lockboxes.
- both systems use keyed HMACs stored with the data to detect modify attacks.
- all data on the disk is encrypted by the users when it is written. Both systems use symmetric keys, making possible modify attacks where readers collude with storage servers to write data.
- a session key and checksums are used to protect all communication between the server and users.

- keys to lockboxes are distributed by the group servers, and individual user public and private keys are required via a public key infrastructure.
- Cepheus implements lazy revocation, where files are re-encrypted only when they are next updated; SNAD suggests use of a similar scheme.

optimizations

- though different keys encrypt different files in the same group, they are kept in lockboxes locked with the same group key, so users need only one key per group.
- long-term keys encrypt all files.
- both systems use block-level encryption (8 KB blocks for Cepheus, 4 KB for SNAD) to allow updates of the individual parts of larger files.

A recent extension to Cepheus and SFS also assumes untrusted servers, and further seeks to detect attacks by the server on the integrity of stored data [Mazieres01]. For instance, one can detect when the server provides different versions of the same file to different users.

3.5 NASD

Network-Attached Secure Disks (NASD) [Gobioff99a] proposes a distributed network of intelligent disks with a shared group server (that also handles metadata for directory traversals). Access for data objects on the disks is authorized by the group server who hands a capability to the user. The disk and group server share a key, and presented with the appropriate capability, the disk services the request. Data is stored in the clear on the disks, but all communication is encrypted. NASD has the following characteristics:

players

- owners, readers, and writers are differentiated.
- the group server and namespace server is integrated into a single metadata server (the file manager), which is clearly distinct from the storage servers.

trust assumptions

- all messages on the wire are encrypted.
- since data is stored in the clear on the storage servers, NASD is vulnerable to attacks in collusion with the storage server.
- since all authentication and authorization data is present in the metadata server, NASD is vulnerable to attacks in collusion with the metadata server.

security primitives

- the metadata server authenticates and authorizes clients by handing them capabilities, which are later verified by the storage server.
- data is encrypted on the wire, and integrity is guaranteed using a MAC on checksums.
- the centralized metadata server makes revocation fast.

trust assumptions

- owners delegate capability distribution to metadata servers. The storage and metadata server are assumed to be trusted; all data is stored in the clear.

granularity

- checksums and keyed MACs ensure the integrity of requests and data transfer between clients and servers.
- introduces a scheme of pre-computed checksums for stored data to reduce the computation of generating checksums on each individual request.

NASD for the first time suggests that individual disk drives directly participate in security protocols. This requires at a minimum strong checksums and keyed MACs for integrity, and optionally encryption and decryption for privacy.

3.6 iSCSI

iSCSI [Satran01] is a draft IETF standard to connect hosts to SCSI devices using TCP as the transport. Since devices may be used across the Internet, security is a major concern. There is a draft proposal [Klein00] to implement a security protocol within iSCSI to authenticate hosts and protect the integrity of commands on the wire. The main characteristics of this proposal are:

players

- there is no notion of individual users; readers, writers and owners are all the same as the host on which they operate. The protocol leaves the issue of authenticating and authorizing individual users to the host.
- there is no group or namespace server, only a storage server.

trust assumptions

- although the storage servers and hosts are mutually authenticated, data is not protected from the server; making it vulnerable to attacks involving collusion with the server.

security primitives

- servers and hosts authenticate using a public and private key mechanism.
- the server does not explicitly differentiate between reads and writes.
- data and commands are encrypted while on the wire using IPsec [Kent98].

- the key for authentication is distributed by an external mechanism.
- revocation is achieved by changing the access control list.

granularity

- session keys are negotiated on a per-login basis.

3.7 LUN security

Disk arrays aggregate the individual disks in the array into logical units (LUNs), which are then accessed by host systems through a host bus adapter (HBA). LUN security proposes to control the access of particular LUNs from different HBAs. This is facilitated by unique IDs on the HBAs and world wide unique numbers which identify them. The host operating system and device driver are trusted not to forge or spoof IDs.

LUN security can be implemented either at the host [HP01a], in the network switch [Brocade01], or at the storage controller [HP01]. The following is true in general of these solutions:

players

- there is no notion of individual users. One can designate read-only permission to some hosts.
- there is no group or namespace server.

trust assumptions

- all players are trusted to identify themselves correctly. The network and servers are also trusted.

security primitives

- typically, players are identified by their world wide number, and this is used for authentication.
- authorization can be performed by maintaining an access control list as follows:
  - at the hosts, by setting up the set of storage controllers that the host may contact;
  - on the wire, by controlling the port mapping at the network switches
  - at the storage server, by setting up the list of HBAs allowed to access each LUN.
- no encryption is performed on the wire or on disk.
- revocation is achieved by changing the access control list.

3.8 AFS

AFS [Howard88] is one of the first distributed file systems that specifically addressed security issues. AFS assumes untrusted users, and uses Kerberos to authenticate users to servers. At the beginning of a session, users obtain *tokens* from a Kerberos server, which authorizes them to access the storage servers. AFS servers verify the tokens and then do appropriate authorization based on group information maintained by a group server. A secure version of RPC is used to protect communication, though some questions have been raised regarding this [Gobioff99a]. The key characteristics of AFS are:

players

- AFS servers act both as storage servers and group servers; authentication is performed by a separate Kerberos server.
- readers, writers, and owners are differentiated based on access control lists at the storage server.

trust assumptions

- apart from the users and the network, all other players are assumed to be trusted. AFS is vulnerable to leak, modify, and destroy attacks in collusion with any of the servers.
- if a user's group information is changed (or revoked) the user continues to have access to files in that group until the user's token expires.

security primitives

- Kerberos authentication is used.
- the servers maintain per-directory access control lists to authorize accesses. The underlying UNIX file permissions are also applied locally.
- AFS does not encrypt on the disk, but RPC messages are secured.
- revocation is done by either changing the access control list or making appropriate changes in the Kerberos server.

security primitives

- though the authentication is centralized, authorization is distributed to the storage servers.
- as in UNIX, users groups are used to simplify authorization rules.

convenience

- single password login via Kerberos, tokens cached for 24 hours by default, often set shorter (e.g., 1 hour) for administrative accounts, or longer (e.g., 30 days) for long-running applications.

3.9 NFS

There have been a number of proposals to build a secure networked file system by providing a security layer on top of NFS. These include proposals to secure the RPC [Taylor86] and tunnelling NFS through SSH or SSL [Gerraty99] to protect data on the wire. The security assumptions and implications of these systems closely match those of AFS and NASD.

The recent NFSv4 specification [Shepler00] explicitly addresses the problem of securing the RPC mechanism. Currently it proposes at least three security mechanisms: one using Kerberos and two using a public key infra-

structure. All these essentially set up a secure communication channel and enable mutual authentication. Interestingly, one of these mechanisms, low infrastructure public key mechanism - exploits the fact that the client authentication can proceed after establishing a secure channel, to reduce the PKI overhead. In this scheme only the server needs to have a public/private pair which authenticates the server and sets up a secure channel. NFSv4 also greatly expands the use of ACLs for access control, very similar to AFS ACLs.

3.10 Windows EFS

The Encrypting File System (EFS) for Windows is integrated into NTFS and supports securing data similar to CFS [Microsoft99]. To facilitate file sharing, EFS uses lockboxes to hold the key of the encrypted file. This lockbox contains the file encryption key protected by a public/private key. EFS supports key escrow by including a key recovery agent among the users allowed to access any file. EFS encrypts and decrypts data just prior to the disk, so some external network security solution is required to secure the data on the wire to a remote server. The characteristics of Windows EFS are as follows:

players

- owners, readers, and writers are differentiated.
- the operating system functions as the group, storage, and namespace server.

security primitives

- Windows primitives are used for authenticating and authorizing writes.
- data is stored encrypted on the disk.
- data is sent in the clear on the wire.
- a user's private key is used to get the file encryption key. Some external mechanism must exist to distribute users' public keys.
- revocation requires re-encrypting files with a new encryption key and re-encrypting the lockbox.
- revocation is achieved by changing the access control list

trust assumptions

- EFS is vulnerable to attacks on the wire if used without an external secure network solution.
- EFS secures against leak and modify attacks mounted in collusion with the server.
- if the private key of the key recovery agent is compromised, all files in the system are protected only by the server's authentication and authorization primitives.

optimizations

- user groups are used by the native Windows file access control lists.
- files are encrypted using a long-term key.

3.11 Survivable storage

Addressing destroy attacks in collusion with storage servers requires survivable storage, i.e., some mechanism to recover from the total loss of a storage server by keeping multiple copies of the data. Several projects currently underway attempt to address security and long-term protection on a much wider scale (in space and in time) than any existing system. PASIS considers storage where data integrity is maintained in the face of the destruction or compromise of some number of replicas [Wylie00] and OceanStore considers a world-wide set of encrypted replicas [Kubiatowicz00]. Another mechanism for protecting data from unauthorized modifications is to use versioning on the storage servers so that data can be reverted to a state before an intrusion, as proposed by S4 [Strunk00]. The most powerful system to protect against all types of destroy attacks might well use a combination of these two schemes, as Carnegie Mellon has proposed by using S4 as a file system on top of PASIS storage.

4 Evaluation

This section explores the costs of implementing the various design choices discussed above, and the impact of these choices on security. The purpose of presenting this data is to compare the relative costs of the systems discussed in Section 3 using a trace from a real system. This allows us to evaluate expensive operations such as full-bandwidth encryption, key distribution, and key generation in practice.

The basis for our evaluation is a 10-day trace of all file system accesses done by a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage

	12-hour	*10-day*
hours	12	240
requests	11.5 million	97.4 million
data moved	23 GB	129 GB
active users	23	32
user accounts	207	207
active files	111,000	969,000
total files	4.0 million	4.0 million
file systems	24	24

Table 3. *Overview of file system trace used for evaluation.* The 10-day trace covers a period in late 2000 from a Thursday to the following Saturday. The 12-hour subset covers 8am to 8pm on the first trace day.

space. The trace was collected by instrumenting the kernel to log all file system calls at the syscall interface. Since this is above the file buffer cache, the numbers shown will be pessimistic to any system that attempts to optimize server messages or key usage based on repeated access. Table 3 provides an overview of the trace.

Implementing each of these systems in the same environment, with the same users, in order to perform a controlled experiment would be prohibitively expensive. We use an analysis of the trace to estimate how the system would behave and compare the relative operation costs. This requires us to make some inferences about the design of the various systems that are not always specified – we highlight these assumptions when they might affect the comparison.

4.1 Security primitives

Table 4 shows the total number of cryptographic operations required for particular security primitives, depending on the granularity at which they are implemented. This clearly illustrates the difference between the on-the-wire and on-the-disk encryption systems. In NASD, the server bears the cost of both the checksums and the encryption (assuming the privacy security level). This cost is reduced somewhat by the pre-computed checksums, but the encryption cost remains high. Since a session key is computed for each client/server interaction, the same file sent to multiple clients must be encrypted each time. In CFS, on the other hand, data is encrypted by the clients before ever being sent to the server. This provides the same level of privacy when data is on the wire, but requires only checksums and signatures at the server, as shown for Cepheus.

4.2 Granularity of protection

The primary comparison among the encrypt-on-disk systems is the level of protection and complexity of key management, and how keys are aggregated to objects.

Table 5 gives counts for the total number of keys used in each of the three high-level classes of designs – using per-session keys, per-file keys,or per-group keys. The table shows the number of keys on a per-user basis for several representative users and system userids during the 12-hour trace period. The representative usernames listed include the busiest users in terms of key use and key distribution, as well as several system userids that own substantial numbers of files. The first three columns consider per-session keys as used in the encrypt-on-wire systems. The middle four columns consider per-file keys as a logical extreme. The last four columns consider a per-group key scheme such as that used in Cepheus. The table shows the number of keys each user would need to obtain during the trace period if keys were created only for each *permission group* of files (i.e., where all files that have the same owner, group, and UNIX permissions bits share a single key). We see that the number of keys

		total ops (10 days)		*peak load (1 minute)*		*NASD*	*CFS*	*SFS*	*Cepheus*
	operation	*messages (1,000s)*	*bytes (MB)*	*messages (req/s)*	*bandwidth (MB/s)*				
server - integrity	message signatures	97,400	n/a	6,600	n/a	X	–	X	X
	checksums	37,300	129,000	6,600	12.5	–	–	X	–
	pre-computed cksums	14,600	n/a	1	n/a	X	–	–	X
server - privacy	encryption (reads)	22,700	77,700	780	6.4	X	–	–	–
	decryption (write)	14,600	51,400	740	6.1	X	–	–	–
client - privacy	encrypt/decrypt	37,300	129,000	1,520	12.5	X	X	X	X
server - key exchanges	per request	97,400	–	6,600	–	–	–	–	–
	per open/close	7,700	–	433	–	X	–	X	X
	per logical volume	24	–	4 / min	–	–	X	–	–
group server - key distribution	per file	11,100	–	1,100	–	X	X	–	–
	per group	177	–	18	–	–	–	–	X
	per logical volume	24	–	1	–	–	–	X	–

Table 4. *Number of cryptographic operations at the server for each design.* The total number of cryptographic operations performed by the server over the course of the 10-day trace, and during the busiest 1 minute interval in the trace. Message signatures are calculated for every request, checksums only for READ and WRITE requests. Checksums and encryptions/decryptions have a per-byte cost, whereas key exchanges and distributions do not. When using pre-computed checksums, only WRITE operations incur server checksumming. The peak load in terms of messages is an interval filled almost entirely with STAT requests; the peak load in terms of bytes has a much smaller number of READ/WRITE requests. The main cost difference can be seen in the privacy rows. In the encrypt-on-wire systems, both server and client work is required, whereas the encrypt-on-disk systems do not require the server work. The granularity chosen for keys has a large effect on the number of messages required for key setup and for key distribution by the group server, as shown in the last six rows. The values in the peak load column give the total streaming and per-message performance required from the server and client processors, or by any hardware engine that might offload the cryptography. The final four columns specify which systems bear which costs; an "X" means that the system uses the indicated cryptographic operation.

	session keys			per-file keys				per-group keys			
user	per request	per open/close	per filesys/lv	total	owner	non-owner	newly created	total	owner	non-owner	newly created
alice	85,100	6,130	6	203	6	197	85	29	1	28	0
bob	8,200	755	4	22	3	19	7	18	2	16	0
charlie	158,000	10,400	5	429	32	397	120	49	6	43	0
dick	46,500	2,360	3	475	62	413	3	24	8	16	0
erik	1,450,000	44,200	7	1,060	571	486	46	43	10	33	0
root	16,000,000	681,000	17	109,000	16,700	92,600	10,400	756	75	681	0
news	1,670,000	264,000	3	103,000	103,000	42	79,200	13	6	7	0
others	945,000	57,100	15	2,300	1,620	684	219	221	84	137	0

Table 5. *Key use by readers and writers.* The number of keys needed if encryption is done on a per-session basis using three different definitions for session: a session per request, a session per open/close pair, and a single session per file system or logical volume (as in NASD, SFS, and iSCSI); on a per-file basis (as in CFS); and on a per-group basis (as in Cepheus). The total number of per-file or per-group keys by username is separated into the total keys used, the number of those keys owned by the user, the number that would have to be obtained from another owner, and the number of new keys created. The row for "others" contains the totals for the thirteen additional usernames active during the 12-hour trace. The rows for usernames "wilkes", "frank" and "bin" that appear in the following table are ommitted here since those users were not active during the 12-hour trace and the columns read 0 across the entire row.

	per-file		per-group	
user	files owned	keys distributed	groups owned	keys distributed
wilkes	54,500	7,810	28	18
alice	19,400	31	13	5
bob	216,000	6,210	17	11
charlie	4,020	148	12	8
dick	13,700	114	13	9
erik	133,000	1,650	14	8
frank	64,900	23,000	32	17
bin	191,000	14,800	33	21
root	240,000	644	129	29
news	1,570,000	554	15	5
others	1,430,000	40,400	2,260	601

Table 6. *Key distribution by owners.* Assuming a system that uses per-file keys, how many keys must a particular owner send to other users. The "owned" columns show the totals for all the files or groups that exist in the file system, and the "distributed" columns show the number of keys sent out during the 12-hour trace. The row for "others" contains the totals for the approximately 200 additional usernames on the system.

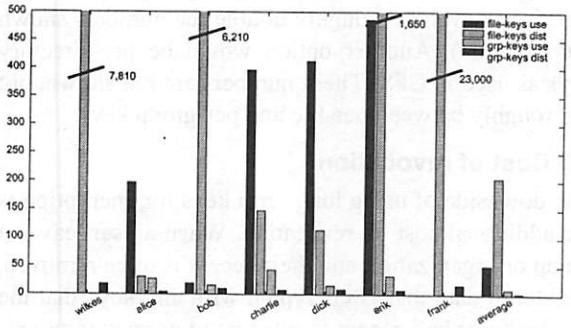


Figure 1. *Per-file vs. per-group keys.* The data of Table 5 and Table 6 presented graphically for several users. Using per-group keys dramatically reduces both the number of keys used by readers and writers and the number of keys that must be distributed by owners. Here "average" is the per-user mean of the "others" rows from the tables.

required for the per-group scheme is orders of magnitude lower than for the per-file scheme and several orders of magnitude less than most of the per-session schemes.

Considering the complexity for owners, as opposed to readers and writers, Table 6 looks at the number of keys that would have to be managed by data owners using per-file or per-group keys. The table shows the total number of keys needed by each owner. The "owned" column gives a count of all the files in the entire file system owned by the given user. The "distributed" numbers show the number of keys a given owner would have had to distribute during the time of the trace to readers and writers of the files for which they are responsible. We can see from these numbers that a system requiring direct user involvement for key distribution would be prohibitively cumbersome (imagine writing 7,500 keys from a possible list of 50,000 on scraps of paper in the course of several hours at your desk).

The two columns on the right are much more promising. They show the number of keys required if we move to a key-per-permission-group scheme. In this case, there is not a separate key for each file, but a key for each class of files, as described above. This produces a much more manageable list with roughly 30 keys per owner, with 10 or 15 of them distributed during a 12 hour period, something that could even be done manually (using scraps of paper) for maximum security. A graphical representation of the difference is shown in Figure 1 where the potential benefit of group keys is clear. An order of magnitude less keys are required for the per-group scheme.

	granularity	files	bytes	NASD	CFS	SFS	Cepheus
aggressive revocation	per-file	3,700	2 GB	–	X	X	–
	per-group	546,000	91 GB	–	–	–	–
lazy revocation	per-file	470	1/2 GB	–	–	–	–
	per-group	121,000	43 GB	–	–	–	X
encrypt-on-wire	per-session	969,000	129 GB	X	–	–	–

Table 7. *The cost of revocation for each design.* Note the explosion in the number of files that must be revoked when a per-group system is used. Aggressive revocation assumes that all affected files are re-encrypted immediately. Lazy revocation assumes that files are re-encrypted only the next time they are read or written, so the values show how much of the data had been re-encrypted after 10 days. This number would increase over time, eventually closing the window of vulnerable data and reaching the aggressive values. Note that even the aggressive, per-group scheme still performs less total encryption work than the encrypt-on-wire scheme which is constantly changing the keys. The final four columns specify which systems bear which costs, an "X" means that the system uses the indicated mechanism.

Note that the numbers in the table are skewed high since our analysis assumes users do not already have any keys cached when the trace starts. In practice, or in a longer trace, the number of keys to be distributed each day would be even lower (e.g., when we consider the entire 10-day trace, the total number of per-group keys distributed is, on average, roughly double the numbers shown for 12 hours). Another option would be per-directory keys as used in CFS. These numbers are not shown, but fall roughly between per-file and per-group keys.

4.3 Cost of revocation

The downside of using long-term keys for encryption is the additional cost on revocation. When a user leaves a group or organization and their access is to be removed, the stored data that is encrypted with any keys that the revoked user had access to must be re-encrypted to prevent future unauthorized access. Table 7 gives details on the cost of revocation when a user leaves a group. In a system that uses the same key for a group of files based on ownership or permissions, there is an additional revocation that results when a user changes permissions on a file (e.g., using *chmod* in UNIX), revocation for this reason is rare in our trace and not covered in the table.

We simulate revocation in our 10-day trace as follows. We choose a single user that will be revoked during this period[1] and track all the keys obtained by this user over the 10-day trace. For aggressive, per-file re-encryption, the number of files re-encrypted is simply all the files the revoked user accessed in the past 10 days. In a system with per-file keys, this is the total amount of data that must be re-encrypted. For a system with per-group keys, the cost includes the re-encryption of all the files in all the file groups to which the user had access. For lazy revocation systems, we assume that file data is re-encrypted as it is read from or written to the system. Data is re-encrypted and re-written whenever the file is accessed for read or write. These values are shown in the lazy revocation portion of the table.

For the lazy revocation scenario in Table 7, the volume of data to be re-encrypted is nearly the same as the work done by an encrypt-on-wire scheme (the server encrypt/decrypt lines from Table 4). This gives further evidence for the duality between encrypt-on-the-wire and encrypt-on-disk schemes. In the encrypt-on-the-wire systems, data is encrypted and decrypted each time it crosses the network. In the encrypt-on-disk systems, data is already encrypted and requires no further work by the server. However, on revocation, the encrypt-on-disk system requires extensive re-encryption. With lazy revocation, this re-encryption occurs whenever the file is read or written, which makes the work done almost comparable to the encrypt-on-the-wire system. The only remaining difference is because encrypt-on-disk needs to perform the encryption only once (until the next revocation), whereas encrypt-on-wire repeats the encryption and decryption each time a file is transferred. The cost differential between the two systems will come down to the relative frequency of revocations, and the total amount of data a particular revocation affects.

5 Conclusions

This paper has developed a common framework of the core functions required for any secure storage system. We have reviewed all the previously proposed systems for storage security, and mapped them into this set of components and design choices. For integrity of network communication, any secure storage system must provide some variant of signed message checksums that strongly tie particular data to particular players. For privacy and confidentiality, we have shown that the two main classes of systems previously described are actually very similar: encrypt-on-wire (which solely protects the communication between servers and users) and encrypt-on-disk

[1] We believe that a frequency of one revocation in 10 days is reasonable. The turnover rate at Silicon Valley companies in the late 1990s averaged around 18% per year, which means that in a group of 200 people, a person would leave about every 10 days.

(which perform encryption and decryption only at user endpoints, with untrusted servers in between). The latter systems provide a form of *pre-computed encryption* for optimizing the encryption work done by the former systems. We have also shown that encrypt-on-disk systems with lazy re-encryption begin to have comparable encryption and decryption costs to encrypt-on-the-wire systems, even though these would seem to be completely different approaches at first glance.

We have quantified the costs of the various systems using a trace from a UNIX timesharing server and shown that the choice made about granularity of keys greatly affects both the complexity and encryption load – sometimes by orders of magnitude. We have quantified a number of design choices that affect security and performance: owner-based key distribution, precomputed encryption, the use of file groups, and lazy re-encryption. We have briefly mentioned survivable storage systems, but not analyzed their performance.

Our experience describing this framework has helped us focus our thinking on to how to build a comprehensive secure storage system that allows users to trade off their level of security and system performance in a concrete, sensible way. Our future work will follow these design choices and a sequel to this paper will report on a system that we are currently developing.

Acknowledgements

We thank all the users of SSP for allowing us to trace their behavior and report on it here. We thank Qian Wang and Jay Wylie for reading earlier drafts of this paper and asking questions we hadn't considered. We thank the anonymous reviewers, and above all our shepherd, Erez Zadok, for his tireless support, encouragement, and attention to detail; including the discussion in Section 2.5, which we would have forgotten about.

References

[Adams99] A. Adams and M. Sasse. Users are not the enemy: why users compromise security mechanisms and how to take remedial measures. *CACM* 42 (12), December 1999.

[Biham90] E. Biham and A. Shamir, Differential Cryptanalysis of DES-like Cryptosystems, *Advances in Cryptology (CRYPTO '90)*, August 1990.

[Blaze93] M. Blaze. A cryptographic file system for UNIX. *Proceedings of 1st ACM Conferenc on Communications and Computing Security*, 1993.

[Blaze94] M. Blaze, Key management in an encrypting file system, *Summer USENIX*, June 1994.

[Boneh96] D. Boneh and R. Lipton. A Revocable Backup System. *USENIX Security Symposium*, July 1996.

[Brocade01] Brocade Communications Systems, Inc. Advancing Security in Storage Area Networks. *White Paper*, June 2001.

[Cattaneo97] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. *Technical Report, University of Salerno*, 1997.

[Cattaneo01] G. Cattaneo, L. Catuogno, A. Del Sorbo and P. Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. *FREENIX 2001*, June 2001.

[Cravotta01] N. Cravotta. Accelerating high-speed encryption: one bottleneck after another. *EDN*, August 2001.

[Dalton01] C. Dalton and T.H. Choo. Trusted Linux: An Operating System Approach. *CACM* 44 (2), Feburary 2001.

[Fu99] K. Fu. Group sharing and random access in cryptographic storage file systems. *MIT Master's Thesis*, June 1999.

[Fu00] K. Fu, M. Kaashoek and D. Mazieres. Fast and secure distributed read-only file system. *OSDI*, October 2000.

[Gerraty99] S. Gerraty. sNFS: secure NFS. *www.quick.com.au/Products/sNFS.html*, October 1999.

[Gobioff98] H. Gobioff, D. Nagle and G. Gibson. Integrity and Performance in Network Attached Storage. *Technical Report CMU-CS-98-182*, December 1998.

[Gobioff99] H. Gobioff, D. Nagle and G. Gibson. Embedded Security for Network-Attached Storage. *Technical Report CMU-CS-99-154*, June 1999.

[Gobioff99a] H. Gobioff. Security for high performance commodity subsystem. *Ph.D. Thesis, CMU-CS-99-160*, July 1999.

[Gutmann96] P. Gutmann. Secure FileSystem (SFS). *www.cs.auckland.ac.nz/~pgut001/sfs*, September 1996.

[Howard88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West. Scale and performance in a distributed file system. *ACM TOCS* 6 (1), February 1988.

[HP01] Hewlett-Packard Company. hp surestore secure manager xp. *www.hp.com/go/storage*, March 2001.

[HP01a] Hewlett-Packard Company. hp OpenView storage allocator. *www.openview.hp.com*, October 2001.

[Hughes99] J. Hughes and D. Corcoran. A universal access, smart-card-based, secure file system. *Atlanta Linux Showcase*, October 1999.

[Kent98] S. Kent and R. Atkinso. Security Architecture for the Internet Protocol, *RFC 2401*, November 1998.

[Klein00] Y. Klein and E. Felstaine. Internet draft of iSCSI security protocol. *www.eng.tau.ac.il/~klein/ietf/ietf-klein-iscsi-security-00.txt*, July 2000

[Kubiatowicz00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *ASPLOS*, December 2000.

[Liadis00] J. Iliadis, D. Spinellis, D. Gritzalis, and B. Praneel. Evaluating certificate status information mechanisms. *CCS'00*, 2000.

[Maniatis02] P. Maniatis and M. Baker. Enabling the archival storage ofsigned documents. *FAST*, January 2002.

[Mazieres99] D. Mazieres, M. Kaminsky, M. Kaashoek and E. Witchel. Separating key management from file system security. *SOSP*, December 1999.

[Mazieres01] D. Mazieres and D. Shasha. Don't trust your file server. *HotOS*, May 2001.

[Microsoft99] Microsoft Corporation. Encrypting File System for Windows 2000. *www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp*, July 1999.

[Miller02] E. Miller, D. Long, W. Freeman and B. Reed. Strong Security for Distributed File Systems. *FAST*, January 2002.

[NA98] Network Associates, Inc. PGPdisk (formerly CryptDisk). *www.pgpi.org/products/pgpdisk*, 1998.

[Nechvatal00] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti and E. Roback. Report on the Development of the Advanced Encryption Standard (AES). *National Institute of Standards and Technology*, October 2000.

[Power01] R. Power, 2001 CSI/FBI Computer Crime and Security Survey, *Computer Security Issues & Trends* VII (1), Computer Security Institute, Spring 2001.

[Reiher93] P. Reiher, J. Cook and S. Crocker. Truffles – A secure service for widespread file sharing. *PSRG Workshop on Network and Distributed System Security*, 1993.

[Satran01] J. Satran, D. Smith, K. Meth, O. Biran, J. Hafner, C. Sapuntzakis, M. Bakke, M. Wakeley, L. Dalle Ore, P. Von Stamwitz, R. Haagens, M. Chadalapaka, E. Zeidner and Y. Klein. IPS Internet Draft - iSCSI. *www.ietf.org/ internet-drafts/draft-ietf-ips-iscsi-08.txt*, September 2001.

[Shepler00] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler and D. Noveck. NFS Version 4 Protocol. *RFC 3010*, December 2000.

[Steiner88] J.G. Steiner, C. Neuman and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. *Winter USENIX*, 1988.

[Strunk00] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules and G. Ganger Self-securing storage: protecting data in compromised systems. *OSDI*, October 2000.

[Swank93] E. Swank. SecureDrive. *www.stack.nl/~galactus/ remailers/securedrive.html*, May 1993.

[Taylor86] B. Taylor and D. Goldberg. Secure RPC: Secure networking in the Sun environment. *Summer USENIX*, June 1986.

[Whitten99] A. Whitten and J.D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. *USENIX Security Symposium*, August 1999.

[Wylie00] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote and P. Khosla. Survivable information storage systems. *IEEE Computer*, August 2000.

[Zadok98] E. Zadok, I. Badulescu and A. Shender. Cryptfs: A stackable vnode level encryption file system. *Technical Report CUCS-021-98*, 1998.

Enabling the Archival Storage of Signed Documents

Petros Maniatis Mary Baker
Computer Science Department, Stanford University
Stanford, CA 94305, USA
{maniatis,mgbaker}@cs.stanford.edu
http://identiscape.stanford.edu/

Abstract

Documents in digital formats are increasingly becoming a common form of expression for anything from rants and opinions to transaction records and contracts. Archiving such documents for the long term, particularly when their only form is digital, can be very important. Sadly, the principal digital expression of an author's intent, the digital signature, is not fit for long-term archives of documents; signing keys can expire or become compromised, rendering the documents they signed indistinguishable from illicit forgeries. We propose KASTS, an extension of traditional archival storage systems that enables the long-term storage of signed documents. KASTS combines time stamping of signed documents with storage of past signature verification keys. We argue that such an extended archival storage system is feasible and describe one possible design for it.

1 Introduction

Documents appear in digital form with growing frequency, and some important documents now appear only in digital form. When their intended use is mainly online, as might be the case for a signed public statement, such documents are increasingly stored in online archival repositories, most notably the Web, or in survivable storage systems like Free Haven [10], Freenet [6], Intermemory [13] or OceanStore [18].

To endorse a digital document, that is, to establish the fact that a person believes or promotes the contents of the document, we use digital signatures. As with physical paper-and-pen signatures, digital signatures are required to show the intent of the signer at the time of signing [1].

However, there is a gap between the potential longevity of digital documents and the longevity of the signatures used to endorse them. Many documents, such as service contracts or ownership transfer records, remain valid and useful for a long period of time. Yet the signatures used to endorse them *must* have a short lifespan for at least two reasons. First, secret signing keys can be stolen. Second, older secret signing keys can be recovered computationally by attackers with increasing ease as computers become faster and cryptanalytic methods become more sophisticated.

For both reasons, it is wise to start regarding signatures produced with a given signing key as suspect some time after that signing key was created, depending on the intended use. Therefore, without further support, digital signatures are inappropriate for long-term archives of signed documents; how do we know if the key used to sign a document was actually valid—i.e., still secret and used exclusively by its claimed owner—when the document was signed?

To address this problem, we propose KASTS, an extension of traditional archival systems to accommodate signed documents. The system builds on an idea by Haber et al. [15], by applying the paradigm of notarization to signed digital documents online. KASTS has two components. First, a Time Stamping Service establishes the real time when a digital document is signed. Second, a Key Archival Service allows anyone to request and receive an authoritative record of the appropriate public signature verification key for a signer at any time in the past.

While the fundamental insight of using notarization to preserve signatures is not new, we believe that the contributions of this work—the design of a Key Archival Service and its combination with document time stamping—are novel and help solve one of the most important, still unsolved problems facing long-term archives of signed documents.

In this paper we describe the architecture of KASTS and the functional specification of its components. For clarity, we describe the system in a simplified setting where there is a single, survivable

and globally trusted service of each kind: one Certification Authority, one Time Stamping Service and one Key Archival Service. However, we also describe design decisions, issues and future work seeking to lift the assumptions of uniqueness, immortality and global trust of those services.

Section 2 describes how digital signatures work in the common case and why they are unfit for long-term archives. Section 3 proposes KASTS, a solution to the problem. Sections 4 and 5 give an overview of KASTS from the architectural and functional standpoints, respectively. In Section 6 we detail design considerations for parts of the system we do not build anew, and for the Key Archival Service, which we do design from scratch. Section 7 discusses three thorny deployment issues with KASTS: the meaning of digital signatures, the effects of certificate revocation, and the long-term security of cryptographic constructs. Finally, we present related and future work.

2 The Life Cycle of a Signed Document

In this section we present the overall context into which our system fits. We describe at a high level the steps one must currently take to sign and publish a document, to set and reset signing and signature verification keys, and to verify the signature on a signed document. We use a specific example to clarify the steps and explain why these steps are insufficient for long-term storage of signed documents. The essential problem is that there is currently no way to determine whether a document was signed while the signing key was still valid, or after that key became invalid.

In our example, Jane Grammatical has written a manifesto on "The Societal Perils of Split Infinitives." Jane feels strongly about the subject matter, so she wishes to publish this manifesto online for the benefit of future generations, making sure that the authorship and integrity of the document are never doubted.

As a first step, Jane needs a digital signing facility to sign her manifesto. In public-key cryptography, on which most commercial digital signature schemes rely, signatures are generated and verified with a *signing key pair*. This key pair consists of a secret *signing key*, used to generate digital signatures, and a public *signature verification key*, used to verify signatures produced by the corresponding signing key. To be able to sign digital documents, Jane must first generate such a signing key pair, and then she must publish the signature verification key from her key pair, so that anyone can verify her signatures.

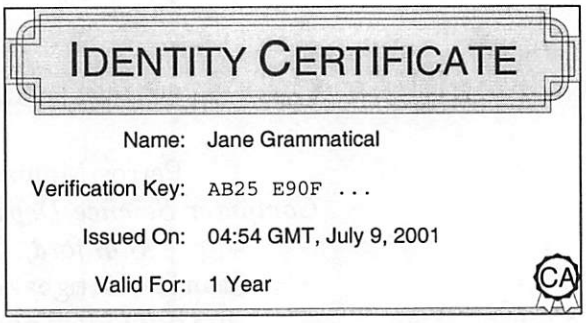


Figure 1: The identity certificate that Jane has been issued by the Certification Authority (CA). It certifies the association of the given verification key with Jane. The key is valid from 7/9/2001 for a year. The icon at the lower right represents the CA's signature on the certificate.

Signature verification keys are published encapsulated within *identity certificates*. An identity certificate is issued by a Certification Authority (CA), such as Verisign, Thawte or Entrust, and certifies the association between an identity name (i.e., an identifier for a signer) and the signature verification key that should be used to verify signatures by that identity. Identity certificates also contain the time at which they are issued and the maximum duration of their validity period. Figure 1 shows a simplified identity certificate for Jane. It indicates that "`AB25 E90F ...`" is Jane's signature verification key for at most a year starting July 9th of 2001. Jane acquires this certificate by contacting the CA securely and sending it her signature verification key. Jane does not, of course, send her secret signing key to the CA or to anyone else; she keeps it hidden and well protected.

To indicate the official character of the certificate, and to protect its integrity, the CA signs every identity certificate it issues using its own signing key pair, also called the *master signing key pair*. Since the CA lies at the top of the certification totem pole and there is no one to certify the validity of its signature verification keys, identity certificates for the CA are different from those issued to Jane: they are signed by the CA itself. CA certificates bootstrap the certification process, which is why they are sometimes referred to as *bootstrapping*, or *root* CA certificates. Because a CA client cannot verify the validity of a bootstrapping certificate since it is self-signed, the CA publishes its certificates via secure channels, for example by postal mail or bundled within store-purchased software, such as web

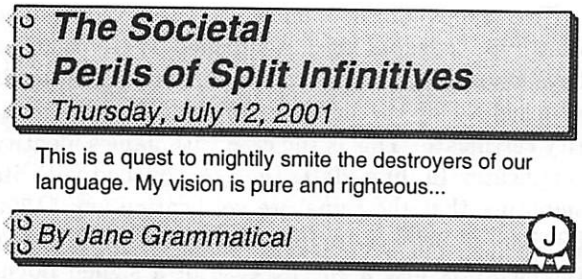


Figure 2: Jane's signed document. The icon on the lower right of the manifesto indicates that the document is signed with Jane's signing key. The signature has been produced by applying the `sign` operation of Jane's favorite digital signing facility with her signing key to the text of the manifesto.

browsers and email clients.

Once Jane has finished proofreading her manifesto, she uses her secret signing key to sign it, producing the signed manifesto of Figure 2. Anyone in possession of the signed manifesto can use the `verify` operation of the digital signing facility and Jane's verification key to check that the document was in fact signed by Jane's signing key.

Jane can now publish her signed manifesto online. As long as her identity certificate is available to readers of the manifesto and there are no security breaches, her authorship of the document is indisputable.

The validity of Jane's signature on the document relies on a "validation chain" consisting of two links. The first link is between Jane and her signing key pair. Unless a verifier knows for a fact that `"AB25 E90F ..."` is Jane's signature verification key, he has no reason to believe that the signature on the manifesto identifies Jane as the signer, even if it is a mathematically correct signature. The second link is between the CA and its master signing key pair. Again, unless a verifier knows for a fact the master verification key, he has no reason to believe that a correct signature on Jane's identity certificate comes from the CA.

Unfortunately, the validation chain can break in two ways (refer to Figure 3 for the relevant timeline). First, any one of the links may become *compromised*; in this scenario, a burglar enters the headquarters of the CA, stealing backup tapes that contain the master signing key, on November 28th, 2001. The break-in is discovered on the following day, so the CA promptly publicizes the event on November 29th. On the same day, a new master signing key pair is generated, and published widely. This burglary breaks the validation chain on Jane's

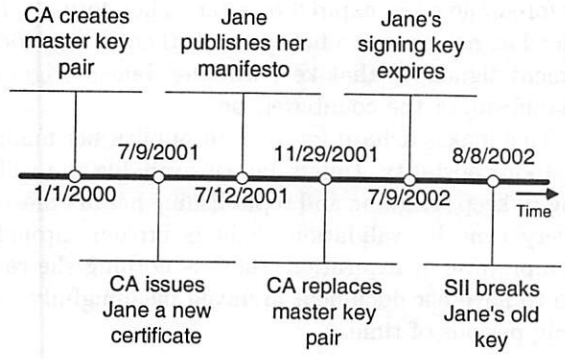


Figure 3: The timeline of the scenario in Section 2.

document since, once the CA's key pair is compromised, it is not clear to a verifier of Jane's signatures whether her certificate was signed by the CA's master signing key before or after November 28th. The burglar could have easily issued new certificates on November 28th, claiming an earlier issuance date; there is nothing anyone can do to distinguish such illicit certificates from legitimate ones.

A second way the validation chain for Jane's manifesto can break is when any one of the links *expires*; in our scenario, Jane's key expires on July 9th, 2002 and the CA's original key would have expired, if it had not been compromised, at the end of its two-year validity period, on January 1st, 2002. One of the reasons that expiration dates are set on identity certificates is to limit the possible amount of damage (i.e., illegitimate signatures produced) that a compromised key can cause, especially if the compromise goes unnoticed. Certificate lifetimes can be set according to the importance of the enclosed key (a master CA key versus the key of a relatively unimportant individual), expected key usage (more signatures mean more fodder for cryptanalysis), and other factors [19].

Sadly, key expiration only compounds the problem for Jane's documents. Once a key expires, all verifiers are expected to assume the key is compromised or "compromisable," and should no longer trust it. In this scenario, Split Infinitives Inc. (SII), a powerful organization favoring the avid use of split infinitives, has devoted large computational resources to discovering Jane's signing key. Since Jane was careful when requesting her certificate, her key expires before SII can possibly recover it. Yet even if SII recovers Jane's signing key *after* July 9th, 2002, they can still write a contradictory manifesto, sign it with the expired recovered signing key and publish it. After Jane's key expires, it is not easy to determine whether the new, illicit manifesto was signed

before Jane's key expired or after. Therefore a verifier has no reason to believe as authentic any document signed by that key, whether Jane's original manifesto, or the counterfeit one.

This makes it hard for Jane to publish her manifesto for posterity. Unless Jane is available and willing to keep resigning and republishing her manifesto every time its validation chain is broken through compromise or expiration, there is nothing she can do to have her document archived meaningfully for long periods of time.

3 Time Stamping Digital Signatures

In this section we explain how combining time stamping and the storage of old signature verification keys helps solve the problem described above.

Time stamping allows a signer to create a proof of when he signed a particular item. In general, the purpose of time stamping is to build a timeline of documents. This is done by a Time Stamping Service (TSS), a trusted but *accountable* third party whose function is to maintain and be able to prove temporal ordering relationships among submitted documents. A time stamp for a document contains the time when the document was stamped and a proof that the document was in fact stamped then. A verifier can check the veracity of a time stamp on a document with the help of the TSS, by checking that the included proof matches the claimed time. Accountability means that, although the service is generally trusted, it can be caught if it "cheats." Cheating in the case of a time stamping service amounts to post-dating, pre-dating, or forgetting about a document. We present how time stamping services are designed in more detail in Section 6.1.

The main idea that helps us solve the problem described in the previous section is to time stamp a signature at the time it is produced [15]. Now a verifier can know whether a signature was generated before or after the event that breaks the validation chain of that signature, such as a discovered compromise or a certificate expiration.

However, time stamping by itself is not sufficient. A verifier who seeks to check the authorship of Jane's manifesto, long after the signing key pair she used has changed, needs to find the appropriate signature verification key. Consequently, we also need some method to archive and retrieve old signature verification keys to enable the long-term archival storage of signed documents.

Two types of keys must be archived. The first type consists of CA-certified keys, that is, keys whose association with a particular identity is vouched for by the signature of the CA on an identity certificate. This is the case with Jane's identity certificate: In July 2001, the CA vouched with its signature that the signature verification key `"AB25 E90F ..."` belongs to Jane. In this respect, Jane's certificate is just a special case of a signed document, and as such, it can be archived in a manner similar to how Jane's manifesto is archived (except for the complications described in Section 7.2).

The second type of key consists of bootstrapping keys, which are traditionally self-certified by the very entity to which they are issued. The master verification key of the CA belongs to this type. A verifier must acquire this key through a secure distribution channel, perhaps by picking it up in person or by receiving it as part of a software distribution. Though this kind of procedure might be practical for obtaining the current master verification key of a unique CA, it can be impractical and unscalable for archived old master keys of perhaps multiple CAs.

For this reason, the need for a Key Archival Service (KAS) becomes clear. This is a trusted, accountable service intended for archiving specifically bootstrapping keys. Nothing precludes conventional, CA-certified keys from being archived at the KAS, but this is not necessary, since conventional keys, encapsulated in identity certificates, can be archived as regular documents.

The system presented in this work, KASTS, extends traditional archival storage systems to accommodate signed documents, using accountable key archival storage and time stamping services.

4 Architectural Overview

Here we present a high-level view of KASTS. Besides the conventional archival storage service it extends, KASTS consists of a TSS, a KAS, and a small client-side library. All certificates are issued by a CA.

The storage service is untrusted, and maintains arbitrary documents submitted to it. KASTS submits signed, time stamped documents, including certificates issued by the CA, to the storage service for long-term storage.

The TSS maintains a timeline of all the documents that it time stamps. It is trusted by everyone within its scope to maintain a unique, tamper-proof timeline, although it remains accountable (see Section 6.1). Anyone who verifies the validity of a time stamp on a document can be convinced that the doc-

ument was signed no later than the time indicated in the time stamp.

The KAS maintains an archive primarily of CA master certificates, but also of any other identity certificates submitted to it. Furthermore, it maintains time stamped snapshots of its archive, with the help of the TSS; in that respect, it is a client of the TSS. It is trusted to maintain a unique, tamper-proof archive, although it remains as accountable as the TSS (see Section 6.2). Anyone who verifies the existence of a certificate in a particular timed snapshot of the KAS can be convinced that the certificate was current and not revoked at the time indicated in the archive snapshot.

Although not a part of KASTS, the CA is an important entity for the system, since it is the issuer of all certificates. It is trusted to maintain a unique, tamper-proof name space at any one time, mapping names to identity certificates.

Clients make use of KASTS via a small client-side library. The interface presented by the library includes the following operations:

1. *publish(identity, document, signature).* The signed document is time stamped and archived. If no associated archived identity certificate for the given identity exists, one is requested.

2. *rekey(identity, new certificate).* The current identity certificate for the given identity, if one exists in the system, is marked as revoked. The new certificate is time stamped and archived.

3. *lookup(identity, time).* The identity certificate associated with the given identity at the given time, if one exists, is returned.

All interactions of the library with the TSS, KAS and CA take place over authenticated and reliable, though not always encrypted channels. As done for CAs, the public keys of the TSS and the KAS are distributed either bundled in purchased software or via other secure media. Interactions of the library or the services with the storage substrate follow the conventions of that substrate; they need not be secured beyond what the substrate itself mandates, since data stored there are self-securing.

In the interest of clarity, we assume the existence of a single TSS, KAS and CA in the remainder of this paper. However, in parallel ongoing work [20], we explore how this design can be ported to a more complex setting where multiple competing TSSes, KASes and CAs coexist. Some of our design decisions are biased by our experiences in that more realistic setting.

5 KASTS in Action

In this section we demonstrate how to use this system, both for publication of signed documents and for later verification of those documents. We explain each of the following steps:

- Following the timeline from Figure 3, on 1/1/2000 the CA publishes a new master verification key, using the process we describe in Section 5.1. This same process is used on 11/29/2001 when a compromise of the master signing key is discovered.

- On 7/9/2001, Jane creates a new signing key pair and, with the help of the CA, a new identity certificate. She then archives the certificate using the process we describe in Section 5.2. She repeats this process on 7/9/2002, when her previous key pair expires.

- On 7/12/2001, Jane signs and publishes "The Societal Perils of Split Infinitives," using the process we describe in Section 5.3.

- On 9/1/2002, a reader wishes to verify the authorship of Jane's manifesto. By that time, both the key pair with which Jane signed the manifesto and the master key pair with which Jane's identity certificate was signed have been replaced. The reader uses the process we describe in Sections 5.4 and 5.5 to retrieve the appropriate old master verification key and then Jane's old identity certificate, respectively, which were current at the time indicated in the manifesto time stamp. With this information, and with the help of the TSS, the reader can now verify the validity of Jane's signature on the manifesto.

5.1 Master Key Storage

The primary objective of this task is to allow the storage of the different master signature verification keys used to verify the CA's signature on individual identity certificates. Every time the CA changes master keys, it updates the key archive, as shown in Figure 4.

First the CA generates a new master signing key pair for itself. It keeps the secret master signing key away from prying eyes, but publishes widely the master verification key (V_{CA} in the figure).

The CA also submits V_{CA}, along with its maximum validity period to the KAS for storage (step 1). Once the storage of the key at the KAS has been

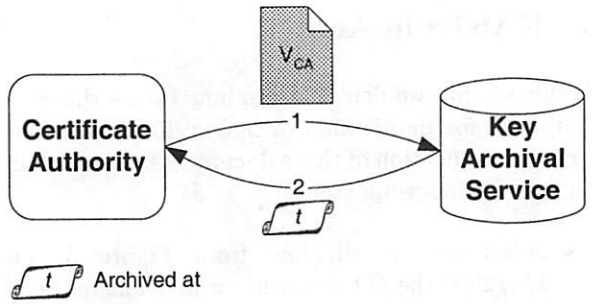


Figure 4: The CA master key storage process, described in Section 5.1.

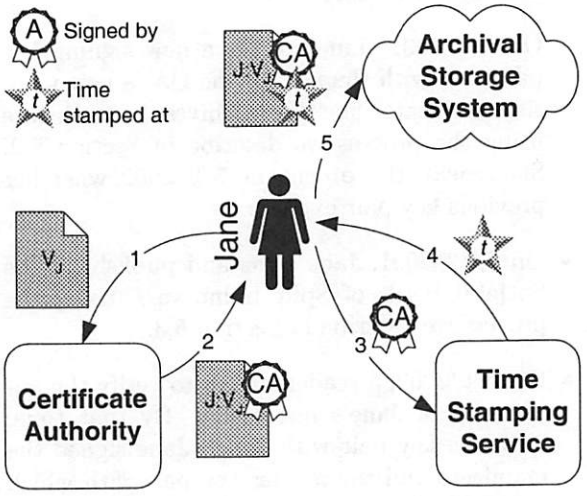


Figure 5: The identity certificate publication process, described in Section 5.2.

completed, the CA may request an optional proof of storage from it. The proof consists of a time stamp of the entire KAS archive after the insertion, and a proof of inclusion of the new key in the archive (step 2). This only serves as an enforcement of the accountability of the KAS. We explain the details of such proofs and the reasoning behind them in Section 6.2.

5.2 Certificate Publication

Jane goes through this process to create and archive her signature verification key. The process is illustrated in Figure 5.

First, Jane generates a new signing key pair. She keeps the secret signing key S_J safe, but submits the public verification key, V_J (or `"AB25 E90F ..."`) to the CA for registration (step 1). The CA returns a new signed identity certificate (marked $J : V_J$ in the figure) to Jane (step 2). This is the certificate shown in Figure 1.

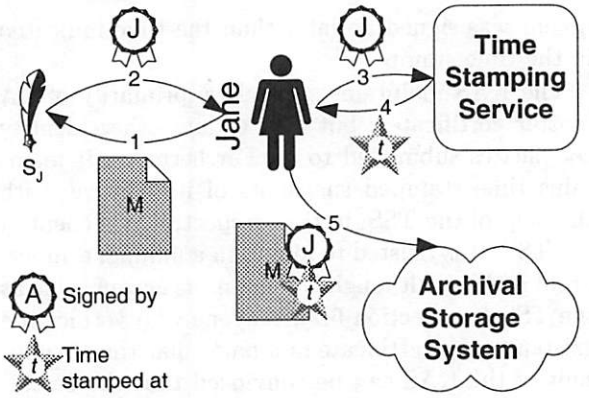


Figure 6: The document publication process, described in Section 5.3.

Jane then submits the CA's signature on her newly acquired certificate to the TSS for time stamping (step 3). The TSS responds with a time stamp on the CA's signature (step 4).

Finally, Jane bundles together her certificate along with the time stamp and publishes them both in the archival storage system (step 5).

It is worth pointing out that, although Jane time stamped the signature of her certificate instead of the entire signed certificate, the result is the same. This is because signatures are cryptographically dependent on the documents for which they are generated, by means of a one-way hash function. Therefore, by time stamping the signature, the TSS effectively also time stamps the entire signed certificate as well.

5.3 Document Publication

Now Jane follows a publication process to place her manifesto in the extended archival storage system. See Figure 6 for an illustration.

First, Jane signs the manifesto (shown as M in the figure) with her secret signing key S_J (steps 1 and 2). She submits the resulting signature to the TSS for time stamping (step 3). Once she receives the time stamp back from the TSS (step 4), Jane submits the bundle consisting of her manifesto, her signature on it, and the time stamp on her signature to the archival storage system (step 5). Again, time stamping the signature is equivalent to time stamping the signed document.

5.4 Master Key Retrieval

To verify the authenticity and authorship of the manifesto, a reader first needs to find the applicable

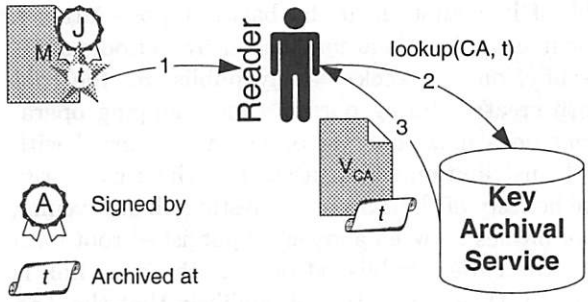


Figure 7: The master verification key retrieval process, described in Section 5.4.

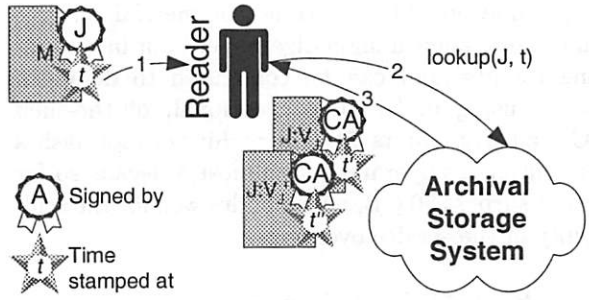


Figure 8: The certificate retrieval process, described in Section 5.5. The inequality $t' \leq t < t''$ holds for the times of the time stamps shown.

master verification key, i.e., the CA signature verification key that was current at the time at which the manifesto claims to have been signed. See Figure 7 for an illustration.

Given the time indicated in the time stamp of the manifesto (step 1), the reader requests a CA verification key from the KAS (step 2).

The KAS returns the applicable master verification key if one is found, along with a proof of its (non)existence there (step 3).

5.5 Certificate Retrieval

Finally, the reader must retrieve the appropriate identity certificate for Jane, given the time at which the manifesto was time stamped, as shown in Figure 8.

The reader searches the archival storage system for identity certificates for Jane around the time at which the manifesto was stamped (step 1). The certificates whose time stamps come immediately before and after the point in time shown in the manifesto time stamp are sought (step 2).

The earlier certificate $(J : V_J)$ is the one whose key is applicable to the signature on the manifesto, and its validity extends until the date of issuance of the later certificate $(J : V'_J)$ (step 3). If no certificate is returned that was time stamped after the manifesto, then the verifier presumes that the maximum duration of the earlier certificate has been used up in full, i.e., he presumes that the key in the certificate was not compromised before the expiration time of that certificate. Section 7.2 discusses some potential complications with this approach and ways to avoid them.

6 Design Issues

In this section we explore the design of the two KASTS components in more detail and we evaluate their viability.

6.1 Time Stamping Service

Centralized TSSes have existed and operated for many years [3, 16, 27]. Their basic functionality allows clients to submit document digests for time stamping at a preset granularity called a *round* (typically one second long) and to submit a time stamped document for subsequent verification. In this section we describe how a TSS works. We use this information to describe how we extend the time stamping model to build a timed archive of keys, in Section 6.2.

The prevalent design for TSSes is based on collision-resistant hash functions [9]. A linking data structure is used to aggregate all document digests submitted for time stamping during the same round. The data structure traditionally used is the Merkle tree [22]. A Merkle tree is a regular k-ary tree, whose contents are all stored in the leaves, sorted using a predetermined total order. Every internal tree node is labeled by concatenating in order the labels of its k children (or *nil* values for missing children) and applying to the result a one-way, collision-resistant hash function. The label of the root is sometimes called the *root hash* of the tree. The root hash "represents" exactly the ordered set of the leaves of the tree. No digest may be added into or removed from the tree without altering the value of the root hash, unless a k-way collision for the hash function can be found, which is believed to be computationally intractable (see Section 7.3 for more details). Figure 9 shows a binary Merkle tree, where $g(.)$ is the hash function, a, b, c and d are the linked data and z the root hash.

A time stamp for a digest consists of the time at which its round was created and a *proof of inclusion* of the digest in the associated linking data

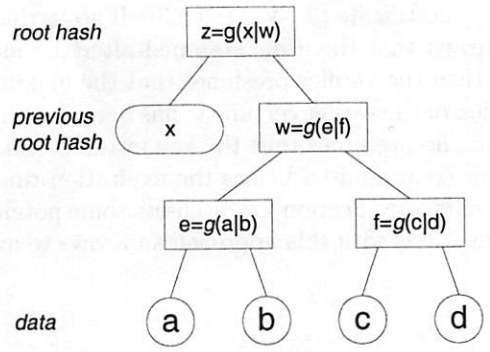


Figure 9: A binary Merkle linking tree containing data a, b, c and d, and the previous round hash x. The concatenation operation is indicated by |.

structure. This proof allows a verifier to determine definitively whether a digest is contained within a linking data structure given the root hash of the structure. Therefore, verifying a time stamp on a document amounts to requesting from the TSS the root hash for the time at which a document claims to have been time stamped and then verifying that the time stamp proves the inclusion of the document digest in the associated linking data structure.

In Merkle trees, the proof of inclusion for a digest consists of all those values that can help recompute the root hash of the tree from that digest. Those values are the labels and locations of the sibling nodes of the digest and of all of its ancestors in the tree. In Figure 9, the proof of inclusion for c consists of the values d, e and x and their locations *right*, *left* and *left*, respectively. Using these, a verifier can compute $z = g(x|g(e|g(c|d)))$, and then compare z to the root hash reported by the TSS for the linking data structure. Assuming that the tree in the figure is created by the TSS at time t, the time stamp for c looks like $[t; right/d, left/e, left/x]$.

A newly created linking data structure depends on the data structure created during the previous second. This dependency is effected by including the root hash of the previous round into the newly created tree. Consequently, a document digest in previous linking data structures cannot change, be added or removed, since that would result in a changed root hash for the associated data structure, which transitively results in a changed root hash for subsequent data structures. In the example of Figure 9, value x is the root hash from the previous time stamping round.

Chaining together linking data structures from older to more recent ones allows TSSes to claim the property of *accountability*. A TSS is accountable if it cannot cheat, by back- or post-dating a document. This is accomplished by periodically—usually, once a week—widely publishing the root hash created during normal time stamping operations on a newspaper or other paper journal with wide distribution. A skeptical TSS client can verify the honesty of the TSS by requesting all intervening root hashes between a given, unpublished root hash and the closest published one. If the hash link is verified, then it is extremely unlikely that the TSS has inserted, modified or deleted digests in its data structures after it published its root hashes.

The feasibility of practical time stamping is no longer questionable. Current commercial services time stamp a few million digests per hour in second-long rounds (and can be configured to do much more), using under 10 conventional, off-the-shelf PC-grade computers [7]. The archive of root hashes for continuous operations of almost a decade so far do not surpass 50 GB, which bodes well for the scalability of the service over time.

6.2 Key Archival Service

The Key Archival Service maintains the timed history of signature verification keys, most notably the master verification keys used and published by the CA, as well as other keys submitted to it for storage. Maintaining this history and making it widely available is essential to the orderly operation of the system we describe here. The functionality of the KAS can be bundled together with the CA or the TSS, although we present it here separately for clarity.

Although the KAS is absolutely necessary only for the storage of the master verification keys of the CA, we have designed it with a much larger data set in mind, for two reasons. First, we want our design to be usable in a more complex setting, where multiple CAs (in the thousands) coexist. Second, we expect that storing non-root keys in the KAS may be advisable, especially given the concerns described in Section 7.2.

In the next three sections we detail the basic data structures used in the KAS, the actual design of the service and, finally, its expected storage complexity.

6.2.1 Data Structures of the KAS

Similarly to the TSS, the KAS accumulates key updates—arriving to KASTS client libraries via *rekey(identity, new certificate)* requests—for a predetermined time period called the *key storage round*. At the end of the key storage round, the archive is

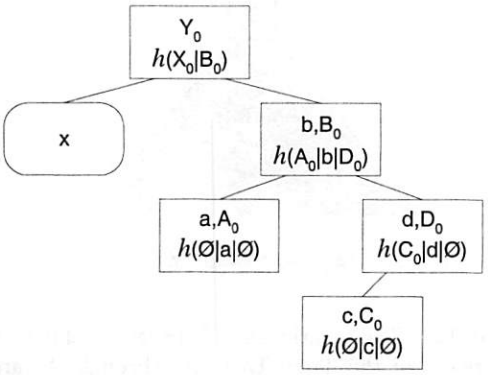


Figure 10: The binary authenticated search tree with the same data as the tree of Figure 9 (rooted at node B_0). Data nodes contain the document digest, and the label of the node. On the second line, the hashing operation that yields the label of the node is shown. $h(.)$ is the hash function. The root hash of the previous tree is x and the root hash for this tree is Y_0. The concatenation operation is indicated by |.

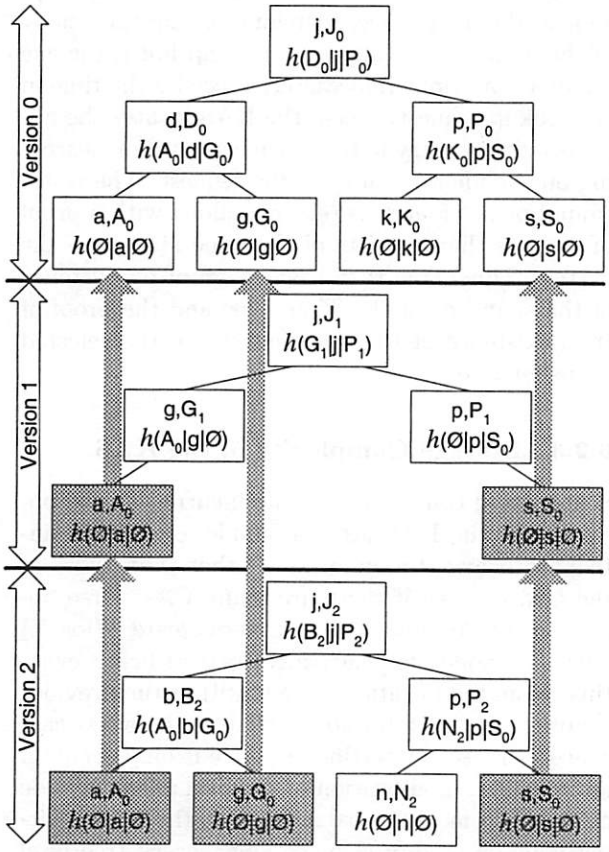


Figure 11: A *versioned, balanced* authenticated search tree. Gray nodes are only references to the original nodes to which gray arrows point. The concatenation operation is indicated by |.

modified and time stamped to reflect the updates that arrived since the previous round, as well as any expirations of previously archived keys. Based on common frequency of key change policies and anticipating the use of the KAS by not only the CA, we currently set the duration of a KAS round to two weeks. Note that durations of TSS and KAS rounds differ by several orders of magnitude.

The simplest design for the KAS would employ a centralized—or centrally administered—database of tuples of the form `<time, identity, verification key, maximum validity period>`. However, to provide the same level of accountability that is offered by the TSS, as seen in Section 6.1, we rely heavily for the design of our KAS on Merkle trees. By using a linking data structure, the KAS can return a proof of storage of the result every time it handles a *lookup* or *rekey* request. The first part of that proof is a time stamp on the particular key storage round of the KAS. The latter part of the proof is a linking tree existence proof of the result in the KAS, similar to proofs described in the previous section.

For the KAS we use a variation of Merkle trees proposed by Buldas et al. [4], called *authenticated search trees*. Buldas et al. suggest this modification to thwart attempts by the maintaining party to keep an inconsistent, unsorted tree linking structure. In authenticated search trees, data occupy not only leaf nodes, but also internal tree nodes. Furthermore, the computation of a node label takes as input the *search key* of the node in addition to the labels of the node's children. The principal contribution of authenticated search trees is that they allow clients who receive an existence or non-existence proof from the tree maintainer to verify that the maintainer is keeping the tree sorted. Figure 10 shows an authenticated search tree containing the same data as the Merkle tree of Figure 9. Again, the root hash of the previous round is x.

Authenticated search trees, like all trees, can be efficiently versioned, so as to preserve different snapshots of the set of stored data without excessive redundancy. Figure 11 shows an example of that. The top tree shows the initial version (version 0) of an authenticated search tree. The middle tree shows version 1, resulting from removing the nodes containing d and k from the tree of version 0. The bottom tree is version 2, which results from inserting nodes containing b and n into the tree of version 1. The grayed out nodes are merely references to the original nodes in version 0, and need not be copied for each subsequent snapshot, unless they change in content or label. Versioning is not useful in the

Merkle trees used in time stamping, since in that case all contents of the tree change entirely from version to version.

Note that in Figure 11, tree operations are *balanced.* This is another welcome property of trees that we use in archiving. Since existence and non-existence proofs have maximum length proportional to the height of the tree, balancing the tree helps such proofs to remain short in the worst case. Balanced trees in our preliminary prototype of the KAS are on-disk B-trees, and are similar to those introduced by Naor et al. [23]. However, no balancing is applied to the topmost level of the tree, so that the previous round hash is always one hash operation away from the new round hash. In Figure 10, only the subtree rooted at the node labeled B_0 is balanced.

6.2.2 KAS Design

There are two kinds of authenticated search trees used in the KAS, *Archive Snapshots* and *Time Trees.* Archive Snapshots store one node for each tuple of the type `<time, identity, verification key, maximum validity period>`, ordered by the identity attribute. There is an Archive Snapshot tree for each distinct version of the KAS archive, i.e., one for each round. Therefore, a single Archive Snapshot contains all the valid keys known to the KAS at the end of the associated round.

Every Archive Snapshot has a distinct root node, as long as it has at least one node difference from the preceding round (we do not alter the archive during rounds with no key updates). This is because inserting, modifying or removing a node results in creating a new version of its parent node, and the changes iteratively percolate up to the root.

The roots of every Archive Snapshot ever built by the KAS are archived in a Time Tree, which is also an authenticated search tree based on B-trees. Time Tree nodes store tuples of the form `<round time, snapshot root>`, ordered by the round time attribute. There is only one current Time Tree within a KAS. At the end of a round, after the new Archive Snapshot is created, a new node for the root of that snapshot is inserted into the Time Tree. See Figure 12 for an illustration of the different trees used in the KAS.

The root G_n of the current Time Tree can be seen as a digest of the history of operations of the KAS up to the end of the previous round, since no Archive Snapshot can change without causing a change to the latest Time Tree root. At the end of round n, G_n is submitted to the TSS for time stamping.

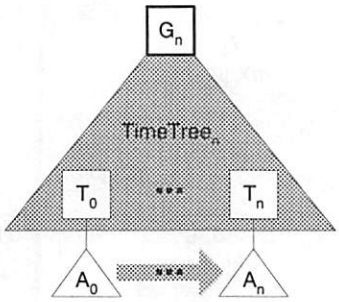


Figure 12: The relationship between Archive Snapshot trees and the Time Tree. A_0 through A_n are all the different Archive Snapshot trees built by the KAS. The thick gray arrow symbolizes the fact that different Archive Snapshots share nodes, as in Figure 11. T_0 through T_n are the corresponding Time Tree nodes, and may be leaves or internal nodes, as per authenticated search trees. The root G_n of the current, n-th ever Time Tree "summarizes" the entire KAS archive.

To respond to a *lookup(identity, time)* request, the KAS first locates the appropriate Archive Snapshot in the Time Tree, searching on the time entry of the request. The appropriate snapshot is the one whose round time immediately precedes the time in the lookup request. Then, the KAS locates the appropriate key entry in the Archive Snapshot, searching on the identity entry of the request. The result (found or not found) is returned along with a proof of storage that consists of the time stamp on the current Time Tree root hash, a proof of existence of the snapshot in the Time Tree and the proof of (non)existence of the returned key in the selected snapshot tree.

6.2.3 Storage Complexity of the KAS

Storage and computation costs incurred by the operation of the KAS are reasonable, even if we anticipate heavy storage of keys other than those of the CA, or even if there are many CAs. Tree operations on Archive Snapshot trees create $\mathcal{O}(\log N)$ new tree nodes for each identity certificate event (insertion, modification or removal), if the previous snapshot had N total nodes. This is the worst case space increase per certificate, since it only occurs if an insertion, modification or removal affects a node at the bottom of the balanced tree, thereby requiring copy-on-modify changes along every tree level to the root. Therefore, the storage required for all Archive Snapshots should be in the worst case on the order of $N \log N$ if a total of N identity certificates are archived. All balanced tree operations take time $\log M$ in the number M of keys in the cur-

rent snapshot, which is much smaller than the total number N of archived keys. Even if every one of the clients of the extended archival storage system uses the KAS to store their signature verification keys, as opposed to storing their identity certificates in the storage substrate, the size of the KAS would still remain in achievable orders of magnitude. An archive of 1 billion verification keys would require roughly 300 TB (assuming minimal disk block fragmentation within B-tree nodes), which is not astronomical for a high-performance service today.

The size of the Time Tree is exactly K nodes, if the KAS contains archives for K distinct snapshots. This, of course, is dependent on the length of the KAS round. Tree operations in the Time Tree take time on the order of $\log K$.

We expect the KAS to receive significantly less traffic than a CA would, for two reasons. First, KAS responses are immutable during a single KAS round, and only change slightly after the passage of each subsequent round in an incremental fashion, to accommodate the increasing size of the Time Tree. Therefore they can be cached very efficiently away from the KAS. In contrast, traffic to CAs usually includes "repeat customers" who check for online certificate revocations. In summary, we expect the long term deployment and operation of a KAS to be at least as feasible as a CA—if not more so.

7 Discussion

In describing KASTS so far we have assumed that a valid signature is one that was time stamped during the validity period of the associated identity certificate. Section 7.1 touches on the distinction between a valid signature and a valid indication of the purported signer's intent.

In Section 7.2 we explain how the use of a conventional storage substrate to store identity certificates can, in some cases, lead to forgery attacks against our system, and we propose a solution.

Finally, in Section 7.3 we describe why we consider time stamping "stronger" than digital signatures.

7.1 Digital Signatures and the Signer's Intent

A fundamental issue that affects what KASTS guarantees and what it does not is the semantic content of a digital signature.

Although real-world, paper-and-pen signatures have enjoyed for centuries often unwarranted absolute trust, digital signatures *do not* establish beyond doubt the identity of the signer. Instead, digital signatures establish beyond doubt whether or not the signer had in his possession a particular secret signing key. The link from a digital signature to the intent of its purported signer is strong only as long as the key used to produce the signature is known exclusively to the individual with whom that key is associated by the CA. The strength of this link has long been considered significantly weaker than that between a paper signature and its signer.

However, digital signatures are slowly becoming legally binding [11]. Although the legal guidelines for their use are fairly specific [1], they have yet to face a significant challenge in court. In the meantime, assuming that the party to whom a signing key is issued bears the liability for anything signed by that key during its validity period, proactive key changes seem to be the only measure against unnoticed key compromise. By changing signing keys frequently and making them short-lived, a signer limits the amount of damage that can be done with any single compromised key.

7.2 No News is Not Good News

In KASTS, all regular identity certificates and other signed documents coexist independently in the same archival storage system. The reason for decoupling a signed document from the identity certificate necessary to verify the signature on that document is efficiency, especially in the case of very short documents. Otherwise redundancy would be unavoidable, since, in general, many documents are signed by the same key.

This means that each complete retrieval and verification of a signed document requires the retrieval of at least two pieces of information from the untrusted storage substrate: the document itself and the *corresponding* identity certificate needed to verify it. The corresponding identity certificate to a signed document is that whose validity period contains the signing time of the document. This validity period is the minimum of the nominal validity period indicated on the certificate itself and the time difference until a newer certificate for the same identity is registered. In other words, a verifier ascribes a validity period shorter than the nominal to Jane's year-long certificate issued on 7/9/2001 once he finds a newer certificate for Jane issued before 7/9/2002.

However, if an adversary has a non-negligible probability of causing individual documents—and therefore individual identity certificates—in the

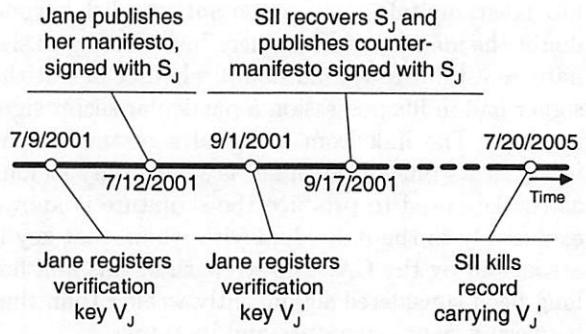


Figure 13: A timeline illustrating how an adversary can kill an identity certificate record and thereby enable the successful verification of a document signed using a compromised signing key.

storage system to disappear or be delayed during retrieval, then he can cause a verifier to consider a signature verification key valid at a time when it was not. Figure 13 illustrates a modification of the earlier scenario from Figure 3. In this scenario, Jane decides to get a new signing key pair on 9/1/2001, long before her old one expires on 7/9/2002. By installing a new identity certificate for herself, Jane essentially revokes the validity of her previous verification key V_J, which would otherwise remain valid until 7/9/2002. Between the issuance of her replacement key pair and the expiration of the old key pair, evil Split Infinitives Inc. successfully recovers her old, now revoked signing key S_J, and uses it to sign and publish a contradicting manifesto. Clearly, this contradicting manifesto should not be considered valid by a verifier, since it is signed after the signing key has been revoked.

Assume now that long into the future, on 7/20/2005, SII manages to kill Jane's newer certificate or temporarily hamper its retrieval. An unaware verifier who retrieves SII's counter manifesto after this time is forced to consider it valid, since he can only find the year-long certificate for V_J whose uncontested validity period contains the signing time of the counter-manifesto.

This attack is dependent on the properties of the archival storage substrate used in the particular implementation. Some systems (e.g., Freenet [6]), while not "trusted" in theory, do have measures to combat directed attacks against specific documents. In the absence of such a storage substrate though, identity certificates should either be stored at a separate storage system offering stronger safeguards against denial of service, be bundled with the associated document at some storage cost, or be decapsulated and stored at the KAS. Given our expected performance characteristics of the KAS, the third option seems attractive.

7.3 Is Time Stamping Unbreakable?

Time stamping digital signatures to extend their lifetime relies on a fundamental premise: that it is harder to break a time stamp than it is to break a digital signature. In fact, this is not trivially true. Time stamping and digital signatures rely on similar cryptographic constructs that build on the conjectured intractability of certain mathematical problems (e.g., discrete logarithms in cyclical groups, factorization of large numbers) or on the conjectured irreversibility of complex one-way algorithms (e.g., the SHA1 hash function [24]). None of these basic building blocks is proven to be secure against arbitrary computational attacks in all realistic settings, even though their security is supported by overwhelmingly strong evidence [21, p. 87].

The difference, however, lies in that digital signatures employ a secret component, a signing key, which can be stolen, leaked, or (with great difficulty) recovered via brute computational force. Instead, the hashing schemes used in most time stamping systems have no such vulnerability, since they do not have a secret key component (also called a *trapdoor*). The only possible attack against such schemes is finding an algorithmic way to annul their computational intractability assumptions.

One technique used to safeguard TSSes against even such groundbreaking attacks is best described as "hedging." Surety, Inc. [27] has patented the practice of concurrently using two different, independent hashing schemes. The hope is that if one of the two hashing schemes is found to have debilitating vulnerabilities, the strength of the other hashing scheme will last until the TSS can take counter-measures, e.g., reissue all time stamps using a new pair of hashing schemes that are still considered impenetrable. The low rate at which computational advances occur against state-of-the-art hashing schemes seems to support the adequacy of this technique.

8 Related Work

Although the basic idea on which KASTS is founded is simple [15], we are not aware of a system design that actually takes advantage of it and works out the details, incorporating both time stamping of signatures and timed storage of old verification keys. The secure archival storage work we are aware

of (for example, OceanStore [18], Cooper et al. [8], SFS-RO [12]) addresses mostly issues of data survivability, format redundancy, confidentiality on-the-wire or on disk, and authentication/authorization of clients and servers. Rosenthal et al. [26] describe a system for preservation of online scientific journals against malicious destruction. The authors voice their concern over the storage of signed documents, but give no definite solutions. We believe that the increasing preference of the business world towards electronic transaction records will only compound the necessity for a design such as ours.

Time stamping seems to be essential to conduct secure transactions with lasting effects in the digital world of the Internet. However, the number of researchers exploring this topic is surprisingly small. Haber and Stornetta [16] introduced the time stamping problem and suggested ways to build linking data structures among documents. Benaloh and de Mare [3], Buldas et al. [5], Goodrich et al. [14] and Naor et al. [23] explored more efficient linking schemes, in the time stamping setting or in that of authenticated dictionaries. A very detailed specification of a time stamping service was produced in the TIMESEC project [25].

Our KAS shares characteristics with earlier work using authenticated data structures, such as Merkle trees and their derivatives. Most notably, the work done by Kocher on the distribution of certificate revocation records [17] relies on this basic idea to distribute certificate revocation records inexpensively. A trusted server creates a binary linking data structure (a Certificate Revocation Tree or CRT) out of all current certificate revocation records. Then, the digest of the tree is distributed securely by that trusted server, but the tree itself is distributed insecurely by untrusted servers. A verifier can always check the validity of a revocation record, as long as a proof of that record's existence in the tree is available and the signed digest of the tree can be retrieved securely. The KAS is more general than the CRT mechanism, in that it maintains timed snapshots of certificates and their revocation or expiration status, to allow the validation of old signatures produced with now-defunct signing keys.

The system we propose is complementary to the basic idea of the Eternity Service [2], for a survivable, incorruptible archive of documents, though intended for a much "tamer" environment. We approach the more hostile environment foreseen by Anderson [2] in future work.

9 Future Work

The basic assumptions throughout this paper are that only one TSS and CA exist, that they are both trusted by everyone, and that they are expected to live forever, as far as the stored documents are concerned. This, unfortunately, is neither practical nor realistic.

Many distinct, competing Certification Authorities exist at the time of this writing. They make revenue out of issuing certificates to their clients and remaining online so as to verify those certificates at a later time. They also capitalize on *reputation*, how much they are trusted to do their job well, and by how many people.

However, CAs are also corporate entities, which may not be trusted by everyone. They must abide by the laws of the land in which they are incorporated and they are staffed by humans who may abide by the same or different laws and who may be coerced to act in ways that do not necessarily parallel the common good, or the good of every single potential client. In that sense, a single CA is not bound to be trusted by everyone in the world. As long as all the participants in a transaction requiring identity certification trust the same CA, all is well. However, in the increasing diversity of electronic transactions, expecting all participants in all transactions to trust the same third party might be considered utopistic. Although fewer commercial TSSes than CAs are in operation today, we expect the same argument to hold for time stamping.

Finally, both CAs and TSSes must obey the laws of business, under which companies come, and very frequently, go. In light of the big market upheaval of the late 1990's and the early 2000's, it would be unreasonably optimistic to assume that any single CA or TSS is going to exist with certainty for a long period of time.

Without the assumptions of CA and TSS uniqueness, global trust and immortality, building a system equivalent to KASTS as described in this paper becomes significantly less straightforward. We motivate, describe and design such a system in [20], using randomized Byzantine fault-tolerant agreement protocols. However, it remains future work to build, evaluate and prove correct a system of such complexity.

10 Conclusions

In this paper we motivate, design and argue for the use of time stamping and timed storage of signature

verification keys to enable the long-term archival storage of signed documents.

The need for time stamping and storage of signature verification keys arises from the inherently short life of digital signatures, especially compared to long-lived documents such as contracts, property titles, transaction records or even works of art.

We design KASTS, an extension to conventional archival storage systems for signed documents, using a Time Stamping Service, and a Key Archival Service that maintains timed snapshots of valid signature verification keys at different times in the past.

We argue that building and operating KASTS is feasible, based on experience with existing TSSes, CAs and archival storage services. In addition, the KAS has low storage requirements (on the order of $N \log N$, where N is the number of different key records being archived) and has expected request rates similar to those of a CA or TSS.

11 Acknowledgments

This work is supported by the Stanford Networking Research Center, by DARPA (contract N66001-00-C-8015) and by Sonera Corporation. Petros Maniatis is supported by a USENIX Scholar Fellowship. We would like to thank Ed Swierk for several close reads of our drafts, Mema Roussopoulos, Henry Tirri and the anonymous reviewers for their helpful comments on this paper, as well as Mahadev Satyanarayanan, our paper shepherd, for his guidance.

References

[1] American Bar Association. *Digital Signature Guidelines.* Jan. 1997.

[2] Anderson, R. J. The Eternity Service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology (PRAGOCRYPT 1996)* (Prague, Czech Republic, 1996).

[3] Benaloh, J., and de Mare, M. Efficient Broadcast Time-stamping. Tech. Rep. TR-MCS-91-1, Clarkson University, Department of Mathematics and Computer Science, Apr. 1991.

[4] Buldas, A., Laud, P., and Lipmaa, H. Accountable Certificate Management using Undeniable Attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS 2000)* (Athens, Greece, Nov. 2000), pp. 9–17.

[5] Buldas, A., Laud, P., Lipmaa, H., and Villemson, J. Time-stamping with Binary Linking Schemes. In *Advances on Cryptology (CRYPTO 1998)* (Santa Barbara, USA, Aug. 1998), H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*, Springer, pp. 486–501.

[6] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, CA, USA, July 2000), H. Federrath, Ed., vol. 2009 of *Lecture Notes in Computer Science*, Springer, pp. 46–66.

[7] Clements, J. Surety, Inc. Personal Communication, July 2001.

[8] Cooper, B., Crespo, A., and Garcia-Molina, H. Implementing a Reliable Digital Object Archive. In *Fourth European Conference on Digial Libraries (ECDL 2000)* (Lisbon, Portugal, Sept. 2000), J. Borbinha and T. Baker, Eds., vol. 1923 of *Lecture Notes in Computer Science*, Springer, pp. 128–143.

[9] Damgård, I. Collision Free Hash Functions and Public Key Signature Schemes. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT 1987)* (Amsterdam, The Netherlands, Apr. 1987), D. Chaum and W. L. Price, Eds., vol. 0304 of *Lecture Notes in Computer Science*, Springer, pp. 203–216.

[10] Dingledine, R., Freedman, M. J., and Molnar, D. The Free Haven Project: Distributed Anonymous Storage Service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, CA, USA, July 2000), H. Federrath, Ed., vol. 2009 of *Lecture Notes in Computer Science*, Springer, pp. 67–95.

[11] Dornin, R. House Passes Digital Signature Bill. *CNN* (Jan. 2000). Available online at `http://www.cnn.com/2000/TECH/computing/01/31/esignatures/`.

[12] Fu, K., Kaashoek, M. F., and Mazières, D. Fast and Secure Distributed Read-only File

System. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, CA, USA, Oct. 2000), USENIX Association, pp. 181–196.

[13] GOLDBERG, A., AND YIANILOS, P. N. Towards an Archival Intermemory. In *Proceedings of IEEE Advances in Digital Libraries (ADL 1998)* (Santa Barbara, CA, 1998), pp. 147–156.

[14] GOODRICH, M. T., TAMASSIA, R., AND SCHWERIN, A. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *2001 DARPA Information Survivability Conference and Exposition (DISCEX 2001)* (Anaheim, CA, USA, June 2001).

[15] HABER, S., KALISKI, B., AND STORNETTA, S. How do Digital Time-stamps Support Digital Signatures? *CryptoBytes, RSA Laboratories 1*, 3 (Autumn 1995), 14–15.

[16] HABER, S., AND STORNETTA, W. S. How to Time-stamp a Digital Document. *Journal of Cryptology: the Journal of the International Association for Cryptologic Research 3*, 2 (1991), 99–111.

[17] KOCHER, P. On Certificate Revocation and Validation. In *Financial Cryptography (FC 1998)* (1998), vol. 1465 of *Lecture Notes in Computer Science*, Springer, pp. 172–177.

[18] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS 2000)* (Cambridge, MA, USA, Nov. 2000), pp. 190–201.

[19] LENSTRA, A. K., AND VERHEUL, E. R. Selecting Cryptographic Key Sizes. `http://www.cryptosavvy.com/cryptosizes.pdf`, Nov. 1999.

[20] MANIATIS, P., GIULI, T., AND BAKER, M. Enabling the Long-Term Archival of Signed Documents through Time Stamping. Technical report, Computer Science Department, Stanford University, Stanford, CA, USA, June 2001. Available at `http://www.arxiv.org/abs/cs.DC/0106058`.

[21] MENEZES, A. J., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] MERKLE, R. C. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy* (Oakland, CA, U.S.A., Apr. 1980), IEEE Computer Society, pp. 122–133.

[23] NAOR, M., AND NISSIM, K. Certificate Revocation and Certificate Update. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX, USA, Jan. 1998), pp. 217–228.

[24] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *Federal Information Processing Standard Publication 180-1: Secure Hash Standard*. Washington, D.C., USA, Apr. 1995.

[25] QUISQUATER, J. J., MASSIAS, H., AVILLA, J. S., PRENEEL, B., AND VAN ROMPAY, B. TIMESEC: Specification and Implementation of a Timestamping System. Technical Report WP2, Université Catholique de Louvain, 1999.

[26] ROSENTHAL, D. S. H., AND REICH, V. Permanent Web Publishing. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track (Freenix 2000)* (San Diego, CA, USA, June 2000), pp. 129–140.

[27] SURETY, INC. Secure Time/Date Stamping in a Public Key Infrastructure. Available at `http://www.surety.com/`.

WOLF–A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems

Jun Wang and Yiming Hu
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030
e-mail: {wangjun, yhu}@ececs.uc.edu

Abstract

This paper presents the design, simulation and performance evaluation of a novel reordering write buffer for Log-structured File Systems (LFS). While LFS provides good write performance for small files, its biggest problem is the high overhead from cleaning. Previous research concentrated on improving the cleaner's efficiency after files are written to the disk. We propose a new method that reduces the amount of work the cleaner has to do before the data reaches the disk. Our design sorts active and inactive data in memory into different segment buffers and then writes them to different disk segments. This approach forces data on the disk into a bimodal distribution. Most data in active segments are quickly invalidated, while inactive segments are mostly intact. Simulation results based on both real-world and synthetic traces show that such a reordering write buffer dramatically reduces the cleaning overhead, slashing the system's overall write cost by up to **53%**.

1 Introduction

Disk I/O is a major performance bottleneck in modern computer systems. The Log-structured File System (LFS) [12, 15, 16] tries to improve the I/O performance by combining small write requests into large logs. While LFS can significantly improve the performance for small-write dominated workloads, it suffers from a major drawback, namely the *garbage collection overhead* or *cleaning overhead*. LFS has to constantly reorganize the data on the disk, through a process called *garbage collection* or *cleaning*, to make space for new data. Previous studies have shown that the garbage collection overhead can considerably reduce the LFS performance under heavy workloads. Seltzer *et al.* [17] pointed out that cleaning overhead reduces LFS performance by more than 33% when the disk is 50% full. Due to this significant problem, LFS has limited success in real-world operating system environments, although it is used internally by several RAID (Redundant Array of Inexpensive Disks) systems [20, 10]. Therefore it is important to reduce the garbage collection overhead in order to improve the performance of these RAID systems and to make LFS more successful in the operating system field.

Several schemes have been proposed [9, 20] to speed up the garbage collection process. These algorithms focus on improving the efficiency of garbage collection *after* data has been written to the disk. In this paper, we propose a novel method that tries to reduce the I/O overhead during the garbage collection, by reorganizing data in two or more segment buffers, *before* data is written to the disk.

1.1 Motivation

Figure 1 shows the typical writing process in an LFS. Data blocks and inode blocks are first assembled in a segment buffer to form a large log. When the segment buffer is full, the entire buffer is written to a disk segment in a single large disk write. If LFS has synchronous operations or if dirty data in the log have not been written for 30 seconds, partially full segments will be written to the disk. When some of the files are updated or deleted later, the previous blocks of that file on the disk are invalidated correspondingly. These invalidated blocks become holes in disk segments and have to be reclaimed by the garbage collection process.

The problem with LFS is that the system does not distinguish *active data* (namely short-lived data) from *inactive data* (namely long-lived data) in the write buffer. Data are simply grouped into a segment buffer randomly, mostly according to their arrival order. The buffer is then

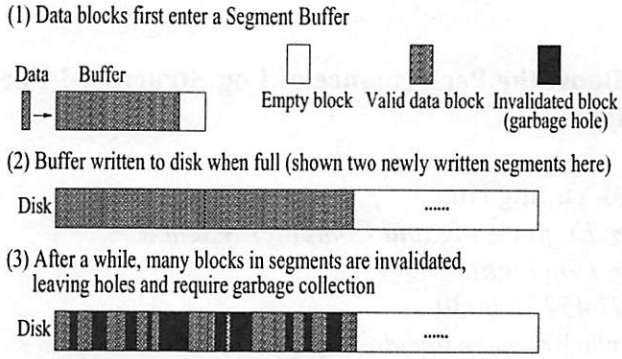


Figure 1: The writing process of LFS

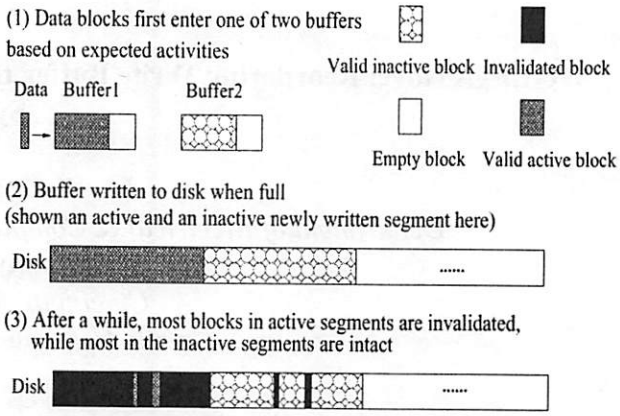


Figure 2: Our new scheme–WOLF

written to a disk segment when it is full. Within the segment, however, some data are active and will be quickly overwritten (therefore invalidated), while others are inactive and will remain on the disk for a relatively long period. The result is that the garbage collector has to compact the segment to eliminate the holes in order to reclaim the disk space.

1.2 Our New Scheme

Based on this observation, we propose a new method called WOLF (reordering Write buffer Of Log-structured File system) that can dramatically reduce the garbage collection overhead. Instead of using one segment buffer, we use **two or more** segment buffers(here is two), as shown in Figure 2. When write data arrives, the system sorts them into different buffers according to their expected longevity. Active data are grouped into one buffer, while less-active data are grouped into the other buffer. When the buffers are full, two buffers are written into two disk segments using two large disk writes (one write for each buffer).

Because data are sorted into active and inactive segments *before* reaching the disk, garbage collection overhead is drastically reduced. Since active data are grouped together, most of an active segment will be quickly invalidated (sometimes the entire segment will be invalidated, and the segment can be reused right away without garbage collection). On the other hand, very few data blocks in an inactive segment will be invalidated, resulting in few holes. The outcome is that data on the disk have a bimodal distribution, namely segments are either mostly full or mostly empty. Similar to Rosenblum and Ousterhout's analysis [15], this is an ideal situation. In a bimodal distribution, segments tend to be nearly empty or nearly full, but few segments are in between. The cleaner can select many nearly empty segments to clean and compact their data into a small number of segments. The old segments are then freed, resulting in a large number of available empty segments for future use. Furthermore, there is no need to waste time to clean the nearly-full segments.

Basically, while previous researchers agreed that the cleaner plays one of the most important roles in LFS, their work focused only on *making the cleaner more efficient* **after** data are written onto the disk. We believe that there exists another opportunity to improve the LFS performance. By re-organizing data in RAM **before** they reach the disk, we could also *make the system do less garbage collection work.* Traditional LFS did try to separate active data from inactive data and force a bimodal distribution, but only during the garbage collection period, long *after* files are written to the disk. Our simulation shows that significant performance gain can be obtained by applying our new method.

1.3 File Access Locality

Accurate prediction of which blocks will be invalidated soon is the key to the success of our strategy. We looked at both the temporal and spatial locality of file accessing patterns. File system accesses show strong temporal locality: many files are overwritten again and again in a short period of time. For example, Hartman and Ousterhout [7] pointed out that 36%–63% of data would be overwritten within 30 seconds and 60%–95% within 1000 seconds in the system they measured. In year 2000, Roselli et al. [14] pointed out that file accesses obey a bimodal distribution pattern: some files are written repeatedly without being read; other files are almost exclu-

sively read. Data that have been actively written, should be put into active segments, and others into inactive segments.

File system accesses also show strong spatial locality, as many data blocks are accessed together. For example, data blocks of one file are likely to be changed together. Similarly, when a file block is modified, the inode of the file, together with the data blocks and the inode of the directory containing the file, are also likely to be updated. These blocks should therefore be grouped together in semantics such that when one block is invalidated, all or most other blocks in the same segment will be invalidated also.

1.4 Related Work

Many papers have tried to improve the LFS performance since the publication of Sprite LFS [15]. Seltzer [16] presented an implementation of LFS for BSD. Several new cleaning policies have also been presented [2, 20, 9]. In traditional cleaning policies [15], including *greedy cleaning* and *benefit-to-cost cleaning*, the live blocks in several partially empty segments are combined to produce a new full segment, freeing the old partially empty segments for reuse. These policies perform well when the disk space utilization is low. Wilkes et al. [20] proposed the hole-plugging policy. In their scheme, partially empty segments are freed by writing their live blocks into the holes found in other segments. Despite the higher cost per block, at high disk utilizations, hole-plugging does better than traditional cleaning because it avoids processing so many segments. Recently, Matthews et al. [9] showed how adaptive algorithms can be used to enable LFS to provide high performance across a wider range of workloads. These algorithms, which use hybrid policies of the above two methods, improved write performance by modifying the LFS cleaning policy to adapt to the changes in disk utilization. The system switches to a different method based on the cost-benefit estimates. They also used cached data to lower cleaning costs. Blackwell et al. [2] presented a heuristic cleaning to run without interfering with normal file access. They found that 97% of cleaning on the most heavily loaded system was done in the background. We proposed a scheme called PROFS which incorporates the knowledge of Zone-Bit-Recording into LFS to improve both the read and write performance. It reorganizes data on the disk during LFS garbage collection and system idle period. By putting active data in the faster zones and inactive data in the slower zones, PROFS can achieve much better performance for both reads and writes [19]. Lumb *et al.* applied a new technique called freeblock scheduling to the LFS cleaning process. They claimed an LFS file system could maintain ideal write performance when cleaning overheads would otherwise reduce performance by up to a factor of three [13].

In this paper, our strategy has a distinctive difference compared with above methods: WOLF works with the initial writes in the reordering write buffers which reduce the cleaning overhead **before** writes go to disk. This scheme finds a new "free" time to solve the same garbage collection problem for LFS. WOLF can be easily combined with other strategies to improve LFS performance. More importantly, it helps LFS provide high performance even in heavy loads and full disks.

Several researchers tried to improve the file system performance without using LFS. Ganger and Patt [4] proposed a method called "Soft Updates" that can eliminate the needs of 95% of synchronous writes. File system performance can be significantly improved because most writes become asynchronous and can be cached in RAM. Hu et al. proposed the Disk Caching Disk [8, 11] which can improve the performance of both synchronous and asynchronous writes. WOLF and Soft-Updates are complementary approaches: The latter improves disk scheduling in traditional file systems through aggressive caching, while WOLF addresses what to do in write caching before the data go to media.

The remainder of the paper is organized as follows. Section 2 describes our design of WOLF. Section 3 describes our experimental methodology. Section 4 shows the simulation results and analysis. Section 5 summarizes our new strategy.

2 The Design of WOLF

2.1 Writing

After the file system receives a write request, WOLF decides if the requested data is active or inactive and puts the write data into one of the segment buffers accordingly. (We discuss how to do this in Section 2.2.) Old data in a disk segment will also be invalidated. The request is then considered complete.

When the write buffers are full, all buffers are written to

disk segments in large write requests in order to amortize the cost of many small writes. Since WOLF contains several segment buffers and each buffer is written into a different disk segment, several large writes occur during the process (one large write for each buffer).

As in the LFS, WOLF also writes buffers to the disk when one of the following conditions is satisfied, even when the buffers are not full:

- A buffer contains modifications that are more than 30 seconds old.
- A *fsync* or *sync* occurs

Since the LFS uses a single segment buffer, when a buffer write is invoked, only one large write is issued. WOLF maintains two or more segment buffers. To simplify the crash recovery process (discussed in Section 2.3), when WOLF has to write data to the disk, *all* segment buffers in RAM will be written (logged) to the disk at the same time. While the logging process contains several large disk write operations since each segment buffer is written to a different disk segment, WOLF considers the log operation *atomic*. A logging is considered successful only if all segment buffers are successfully written to the disk. The atomic logging feature means that we can view the multiple physical segments of WOLF as a *single virtual segment*.

The atomic writing of multiple segments can easily be achieved with a timestamp. All segments written together will have the same timestamp and the same "number of segments written together" field. During crash recovery, the system searches for the segments with the latest timestamp. If the number of segments with the same latest timestamp matches the "number of segments written together" field, then the system knows that the last log-writing operation was successful.

2.2 Separating Active and Inactive data

One of the important problems in the design of WOLF is how to find an efficient and easy-to-implement method that can separate active data from inactive data and put them into different buffers accordingly.

2.2.1 An Adaptive Grouping Algorithm

We developed a heuristic learning method for WOLF. The tracking process implements a variation of the least-recently used algorithm with frequency information. Our algorithm is similar to virtual memory page-aging techniques.

To capture the temporal locality of file accesses, each block in the segment buffers has a reference count associated with it. This number is incremented when the block is accessed. The count is initialized to zero and is also reset to zero when the file system becomes idle for a certain period. We call this period as *time-bar*. It is initialized to 10 minutes[1]. If the age of this block exceeds current time-bar, WOLF will reset the reference count of this block to zero. WOLF only does this zero clearing in write buffers. The value of the count indicates the active level of the block in *most recent active period*, which starts since the time-bar. The higher the value of the count, the more active a block is. The Time-bar could be adaptively tuned for the various incoming accesses. When the system identifies that there is no significant difference among the blocks' active ratios in the reorder buffers, which means the 90% reference counts of blocks are equal, the time-bar will be doubled. If most blocks have too different active ratios, when only 10% reference counts of blocks are equal, the time-bar will be halved. The Time-bar makes the reordering buffers work heuristically for different workloads. Active data are then put into the active segment buffer, and other data in the inactive buffer.

If two blocks have the same reference counts, then spatial locality is considered. If the two blocks satisfy one of the following conditions, they will be grouped into the same segment buffer:

- If the two blocks belong to the same file.
- If the two blocks belong to files in the same directory.

If none of the above conditions is true, the blocks are randomly put into buffers.

The overhead of this learning method is low. Most active blocks have no more than a hundred accesses in a short period. Only a small amount of additional bits

[1] For different workloads, this threshold may be different. We choose this value for most workloads. This threshold works well when active data live less than 10 minutes and inactive data lives more than 10 minutes

are needed for each block. *Time-bar* is managed by the reordering buffer manager with little overhead. WOLF only resets the reference count in the reordering buffers.

2.2.2 Data Lifetimes

In order to choose the proper threshold for different workloads, we calculate the byte lifetime by subtracting the byte's deletion time from its creation time. This "deletion-based" method was used by [1] in which all deleted files are tracked. For considering the effects of overwrites, we measured byte lifetime rather than file lifetime. Figure 3 tells the byte lifetime of four real-world workloads in details(these traces will be described in section 3.2.1).

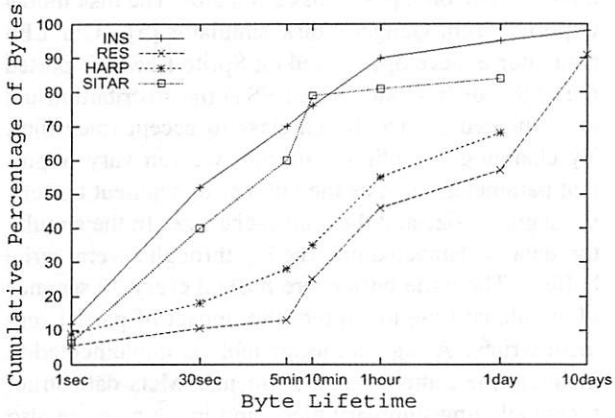


Figure 3: Byte Lifetime of Four Real-world Workloads

From the picture, we can see the active data's lifetimes shows various behaviors in different workloads. More than 70% of the data in INS and Sitar traces have a lifetime less than 10 minutes. Around 35% of the data in RES and Harp traces have a lifetime less than 10 minutes. Since the lifetime of active data varies in different workloads, it is necessary to develop this adaptive grouping algorithm to separate active data and inactive data for different workloads.

2.3 Consistency and Crash Recovery

In additional to LFS' high performance, another important advantage of LFS is fast crash recovery. LFS uses checkpoints and maintains the order of updates in the log format. After a crash, the system only has to *roll forward*, reading each partial segment from the most recent checkpoint to the end of the log in write order, which involves incorporating any modifications that occurred. Thus there is no need to perform a time-consuming job like *fsck*.

In WOLF, data in memory are re-grouped into two or more segment buffers and later written into two or more disk segments. As a result, the original ordering information may be lost. To keep the crash recovery process simple, WOLF employs the following strategies:

1. While data blocks are reordered by WOLF to improve the performance, their original arrival ordering information is kept in a data structure and written to the disk in the summary block together with each segment.
2. While WOLF maintains two or more segment buffers, its atomic logging feature (discussed in Section 2.1) means that these multiple physical buffers can be viewed as a *single virtual segment.*

Since WOLF maintains only a single *virtual segment* which is logged atomically, and the information about original arrival orders of data blocks in the virtual segment is preserved, crash recovery in is nearly as simple as in LFS.

2.4 Reading

WOLF only changes the write cache structures of LFS. The read operations are not affected. As a result, we expect that WOLF has similar read performance as that of LFS when the system is lightly loaded. When the system is heavily loaded, WOLF should have better read performance because of its more efficient garbage collection process that reduces the competition for disk bandwidth.

2.5 Garbage Collection

WOLF does not completely eliminate garbage, therefore garbage collection is still needed. Benefit-to-Cost cleaning algorithm works well in most cases while hole-plugging policy works well when the disk segment utilization is very high. Since previous research shows that a single cleaning algorithm is unlikely to perform equally well for all kinds of workloads, we used an adaptive approach similar to the Matthews' method [9]. This policy automatically selects either the *benefit-to-cost* cleaner or the *hole-plugging* method depending on the cost-benefit estimates.

In WOLF, the cleaner runs when the system is *idle* or disk utilization exceeds a *high water-mark*. In our simulation, the high water-mark is when 80% of the disk is full, and *idle* is defined as the file system has no activities in 5 minutes. The amount of data that the cleaner may process at one time can be varied. In this paper, we allowed the cleaner to process up to 20 MB at one time. To calculate the benefit and overhead of garage collection, we used the following mathematical model. These formula were developed by Matthews *et al.* (See more details in [9]).

The benefit-to-cost ratio is defined as follows:

$$\frac{benefit}{cost} = \frac{(1 - utilization) * age\ of\ segment}{(1 + utilization)}$$

Here utilization represents the ratio of the live bytes to one segment size. Specifically, the cost-benefit values of cleaning and hole-plugging policies are calculated as follows:

$$CostBenefit_{Cleaning} = \frac{TransferTime_{Cleaning}}{SpaceFreed_{Cleaning}}$$

$$CostBenefit_{Plugging} = \frac{TransferTime_{Plugging}}{SpaceFreed_{Plugging}}$$

The adaptive policy always picks up segments with the lower Cost-Benefit estimates to clean. Segments with more garbage (hence very low segment utilization and high benefit-to-cost ratios) will be cleaned first. Older segments will also be cleaned first, as data in younger segments will have a better chance to be invalidated in the future.

Because WOLF's buffer manager separates the active data from inactive data which leads to a bimodal disk segment layout, both the benefit-to-cost and hole-plugging methods can benefit from this nice layout. For benefit-to-cost, since most active segments are mostly garbage (hence very low utilization), their benefit-to-cost ratios are very high. These segments will be cleaned first to yield many blank segments. For hole-plugging, when the adaptive cleaner switches to this method (which will tend to occur in very high disk utilization), cleaner uses the least utilized segments to plug the holes in the most utilized segments. WOLF simply reads the few remaining live bytes from an active disk segment and plug them into the few available slots of an inactive disk segment (very high segment utilization).

3 Experimental Methodology

We used trace-driven simulation experiments to evaluate the effectiveness of our proposed new design. Both real-world and synthetic traces are used during simulation. In order to make our experiments and simulation results more convincing, we use four different real-world traces and four synthetic traces in the comprehensive covering fields.

3.1 The Simulators

The WOLF simulator contains more than 10,000 lines of C++ code. It consists of an LFS simulator, which acts as a file system, on top of a disk simulator. The disk model is ported from Ganger's disk simulator [5]. Our LFS simulator is developed based on Sprite LFS. We ported the LFS code from the Sprite LFS kernel distribution and implemented a trace-driven class to accept trace files. By changing a configuration file, we can vary important parameters such as the number of segment buffers, the segment size and the read cache size. In the simulator, data is channeled into the log through several write buffers. The write buffers are flushed every 30 seconds of simulated time to capture the impact of partial segment writes. A segment usage table is implemented to maintain the status of disk segments. Meta-data structures including summary block and inode map are also developed. We built a checkpoint data structure to save blocks of inode map and segment usage table periodically.

The disk performance characteristics are set in Disksim's config files. We chose two disks for testing, a small (1 GB capacity) HP2247A disk and a large (9.1 GB) Quantum Atlas10K disk. The small HP2247A was used for Sitar and Harp traces, because the two traces have small data-sets (total data accessed < 1 GB). A small disk is needed in order to observe the garbage collection activities. The large disk was used for all other traces. Using two very different disks also helps us to investigate the impacts of disk features like capacity and speed on WOLF performance. The HP2247A disk' spindle speed is set to 5400 RPM. The read-channel bandwidth is 5 MB/sec. Its average access time is 15 ms. The Quantum Atlas10K has a 10024 RPM spindle speed. Its read-channel bandwidth is 60 MB/sec and average access time is 6 ms.

3.2 Workload Models

The purpose of our experiments is to conduct a comprehensive and unbiased performance study of the proposed scheme and compare the results with that of unmodified LFS. We paid special attention to select the traces. Our main objective was to select traces that match as close to realistic workloads as possible. At the same time, we also wanted to cover as wide a range of environments as possible. The trace files that have been selected in this paper are discussed below.

3.2.1 Real-world Traces

Four real-world file system traces we used in our simulation. We got two sets of real-life traces from two different universities to validate our results. Two of them came from University of California, Berkeley, called **INS** and **RES** [14]. **INS** came from a collection from a group consisting of 20 machines located in labs for undergraduate classes. **RES** was attained from 13 desktop machines of a research group. **INS** and **RES** were recorded over 112 days from September 1996 to December 1996. Both traces came from their clusters running HP-UX 9.05. The other set of two traces, from University of Kentucky, contain all disk activities on two SunOS 4.1 machines during ten days for **Sitar** trace and seven days for **Harp** trace[6]. **Sitar** trace represents an office environment while **Harp** reflects common program development activities. More specifically, **Sitar** trace is a collection of file accesses by graduate students and professors doing work such as emailing, compiling programs, running LaTeX, editing files, and so on. **Harp** trace shows a collaboration of two graduate students working on a single multimedia application. Because **Sitar** and **Harp** have a small amount data, we use the small disk model with these two real-world traces. Notice in the experiments, we expand Sitar and Harp by appending files with same access pattern in original traces but with different file names in order to explore the system behavior under different disk utilizations. For large traces with more than 10GB data traffic, we do not use this procedure.

These real-world traces are described in more detail in Table 1.

3.2.2 Synthetic Traces

While real-world traces give a realistic representation of some real systems, synthetic traces have the advantage of isolating specific behaviors not clearly expressed in recorded traces. We therefore also generated a set of synthetic traces. We varied the trace characteristics as much as possible in order to cover a very wide range of different workloads.

We generated the following four sets of synthetic traces:

1. Uniform Pattern *(Uniform)*

   Each file has equal likelihood of being selected.

2. Hot-and-cold Pattern *(Hot-cold)*

   Files are divided into two groups. One group contains 10% of files; it is called *hot* group because its files are visited 90% of the time. The other group is called *cold*; it contains 90% of the files but they are visited only 10% of the time. Within groups each file is equally likely to be visited. This access pattern models a simple form of locality.

3. Ephemeral Small File Regime *(Small Files)*

   This suite contains small files and tries to model the behavior of systems such as the electronic mail or the network news systems. The sizes of files are limited from 1 KB to 1 MB. They are frequently created, deleted and updated. The data lifetime of this suite is the shortest one in this paper (90% of byte lifetimes are less than 5 minutes).

4. Transaction Processing Suite *(TPC-D)*

   This trace consists of a typical TPC-D benchmark which accesses twenty large size database files from 512 MB to 10 GB. The database files consist of the different number of records ranged from 2,000,000 to 40,000,000. Each record is set to 100 bytes. Most transaction operations are queries and updates in this benchmark. The I/O access pattern is random writes followed by sequential reads. Random updates are applied to the active portion of the database. And then sometime later, large sweeping queries read relations sequentially [18]. This represents the typical I/O behavior of a decision support database. In this trace, we use sequential file reads to simulate 17 SQL queries for business questions. As for implementing TPC-D update functions, we generate random writes to represent following categories: updating 0.1% of data per query, inserting new sales data with 0.1% of table size and deleting old sales data of 0.1% of table size.

Features	INS	RES	SITAR	HARP	Uniform	Hot-cold	TPC-D	SmallFS
Data read(MB)	94619	52743	213	520	8000	8000	8000	8000
Data write(MB)	16804	14105	183	249	8000	8000	4000	8000
Read:Write ratio	5.6	3.7	1.16	2.08	1.0	1.0	2.0	1.0
Reads(thousands)	71849	9433	57	26	800	800	800	4000
Writes(thousands)	4650	2216	49	12	800	800	400	4000
File Size(< 16KB)	82.8%	63%	80%	73	80%	80%	0%	95 %
File Size(16KB–1MB)	16.9%	36.5%	19.96%	26.98%	19.95%	19.95%	0%	5%
File Size(1MB+)	0.2 %	0.5 %	0.04 %	0.02 %	0.05%	0.05%	100%	0%

Table 1: Four real-world traces and Four synthetic traces

The other information of these four synthetic traces can be seen in Table 1.

4 Simulation Results and Performance Analysis

In order to understand the insight effect of WOLF, we compare our design with the most recent LFS using adaptive method which is the baseline system. The reason is that we want to explore the effect with our reordering write buffers rather than the adaptive cleaning policy. Therefore, two compared systems use the *same* adaptive garbage cleaning strategy. In experiments, system automatically selects either *benefit-to-cost* or the *hole-plugging* depending on the cost-benefit estimates. WOLF separates active data from cold data to generate active/inactive segments in initial writes. The different disk layouts in two systems lead to different performance.

In our experiments of this paper, we set several default parameters unless specified: a 64 MB read cache, each disk segment is 256 KB and each segment buffer is 256 KB.

4.1 Overall Write Cost

Write cost is the metric traditionally used in evaluating LFS write performance. It only considers the effect of the number of segments. Matthews et al. pointed out segment size also plays a larger role in the write performance. They described a way to quantify this trade-off between amortizing disk access time across larger transfer units and reducing cleaner overhead: *Overall Write Cost*, which captures both the overhead of cleaning as well as the bandwidth degradation caused by seek and rotational latency of log writes [9].

In this paper we used this new metric – overall write cost to evaluate WOLF performance. The following formula are adapted from [9]:

First, two terms, *write cost* and *Transfer Inefficiency* ($Ineff_{Xfer}$) are defined:

$$WriteCost = \frac{Segments\,Transferred_{Total}}{Segments\,Transferred_{NewData}}$$
$$= \frac{SegsW_{NewData} + SegsR_{Clean} + SegsW_{Clean}}{SegsW_{NewData}}$$

Here $SegsW_{NewData}$ is the total number of segments written to the disk caused by new data. $SegsR_{Clean}$ and $SegsW_{Clean}$ are the total numbers of segments read and written by the cleaner, respectively. This term describes the overhead of cleaning process.

$$Ineff_{Xfer} = AccessTime \times \frac{DiskBandwidth}{SegmentSize} + 1$$

$Ineff_{Xfer}$ measures the bandwidth degradation caused by seek and rotational delays of log writes. AccessTime represents the average disk access time.

And finally,

$$Overall\ Write\ Cost = WriteCost \times Ineff_{Xfer}$$

4.1.1 Performance under Different Workloads

In order to understand how the WOLF and LFS perform under different workloads, results for the four synthetic traces and four real-world traces are compared in Figure 4.

It is clear from the figure that the WOLF significantly reduces the overall write cost compared to the LFS. The

new design reduces the overall write cost by up to 53%. The overall write cost is most reduced when the disk space utilization is high. When the disk becomes more full, the garbage collection is more important. WOLF plays the more important role in reducing garbage on the disk.

Although the eight traces have different characteristics, we can see that the performance of WOLF is not sensitive to the variation in workloads. This derives from our heuristic reorganizing algorithm. On the other hand, LFS performs especially poor for the TPC-D workload because of its random updating behavior. This is not a surprise. Similar behavior was observed by Seltzer and Smith in [17]. WOLF, on the other hand, significantly reduces the garbage collection overhead so it still performs well under TPC-D.

4.1.2 Effects of the Number of Segment Buffers with Real-world Traces

Figure 5 shows the results of the overall write cost versus disk utilization for the four real-world traces. We varied the number of segment buffers of WOLF from 2 to 4. We also varied the segment buffer size of the LFS from 256 KB to 1024 KB.

Increasing the number of segment buffers in WOLF would slightly reduce the overall write cost but does not have a significant impact on the overall performance.

The reason we studied LFS with different segment buffer sizes, is to show that the performance gain of WOLF is not due to the increased buffer numbers (hence the increased total buffer size). The separated active/inactive data layout on disk segments contributes to the performance improvement. In fact, for LFS, increasing the segment buffer sizes may actually increase the overall write cost. This observation is consistent with previous studies [9, 15]

Note that because WOLF uses more segment buffers than the LFS does, data may stay in RAM longer. However, this does not poses a reliability problem. As discussed before, in WOLF, if the segment buffers contain data older than 30 seconds, they will be flushed to the disk, just as LFS.

4.1.3 Effects of Segment Sizes with Real-world Traces

The size of the disk segment is also a substantial factor on the performance of both WOLF and LFS. If the size of the disk segment is too large, it would be a little difficult to find enough active data to fill one segment and enough inactive data for another segment. The result will be active data and inactive data are mixed together in a large segment, resulting in poor garbage collection performance. The limited disk bandwidth will also have a negative impact on the overall write cost when the segment buffer size exceeds a threshold. On the contrary, if the segment size is too small, the original benefit of LFS, namely taking the advantage of large disk transfer, is lost.

Figure 6 shows the simulation results with the overall write cost versus the sizes of segment buffers. We can see that for both WOLF and LFS, a segment between 256-1024 KB is good for these kind of workloads.

4.1.4 Segment Utilization Distribution

In order to provide insights into understanding why WOLF significantly outperforms the LFS, we also compared the segment utilization distributions of WOLF and LFS. Segment utilization is calculated by the total live bytes in the segment divided by the size of this segment.

Figure 7 shows the distribution of segment utilizations under the four real-world traces. We can see the obvious bimodal segment distribution in WOLF when compared to the LFS. Results for other workloads are similar. The nice bimodal distribution is the key to the performance advantage of WOLF over the LFS.

4.2 Read/Write Latency

In previous discussion, we used *overall write cost* as the performance metric. Overall write cost is a direct measurement of system efficiency. We have shown that WOLF performs encouragingly better than LFS, as the former has much smaller overall write cost than the latter.

However, end-users would be more interested in user-measurable metrics such as the access latencies [3]. Overall write cost quantifies the additional I/O overhead when LFS does the garbage cleaning. The LFS perfor-

mance is very sensitive to this overhead. To see whether the low overall write cost in WOLF can be translated to low access latencies, we also measured the average file read/write response times in the file system level. We collected the total file read/write latencies and divided the total number of file reads/writes requests. All these results include the cleaning overhead. The results are presented in this subsection.

4.2.1 Write Latencies

Figure 8(a) shows the file write performance of LFS and WOLF under eight traces. Figure 8(b) plots the performance improvement of WOLF over LFS. We can see that WOLF significantly enhances the write performance by 27–35.5%, in terms of improved response times. The lower overall write cost in WOLF directly leads to a smaller write response time. The Hot-cold trace achieves the best improvement because of its good active behavior.

4.2.2 Read Performance

Figure 9(a) shows the file read performance of LFS and WOLF under eight traces. Figure 9(b) plots the performance improvement of WOLF over LFS. The results show that, for most traces, the read performance of WOLF is at least comparable to that of LFS. This is expected, as WOLF does not directly affect the read operations of LFS. Although WOLF changes the physical layout on disk for LFS, WOLF's grouping algorithm includes the similar policy which is used in locality-grouping rules of regular LFS, such that files in same directory are put in same segment and *etc.*, WOLF does not have much impact on the read performance when the load is light. When the load is heavy, we may see a little better read performance of WOLF than that of LFS because WOLF reduces the cleaning overhead so that WOLF ameliorates the competition of disk bandwidth. RES and TPC-D got little loss for their more random reads because random reads have poor spatial locality which results in much longer disk seeks and rotations during garbage collection.

4.3 Implication of Different Disk Models

From the results of sections 4.1 and 4.2, we can see WOLF achieves significant performance gains for both the small/slow and the large/fast disk models. The results suggest that the disk characteristics do not have a direct impact on WOLF. While the absolute performance parameters may vary on different disk models, the overall trend is clear: WOLF can markedly reduce garbage collection overhead under many different workloads on different disk models.

5 Conclusion and Future Work

We have proposed a novel reordering write buffer design called WOLF for the Log-structured File System. WOLF improves the disk layout by reordering the write data in segment buffers *before* writing data to the disk. By utilizing an adaptive algorithm that separates active data from inactive data, and taking advantages of file temporal and spatial localities, the reordering buffer forces actively-accessed data blocks into one hot segment and inactive data into another cold segment. Since most of the blocks in active segments will be quickly invalidated while most blocks in inactive segments will be left intact, data on the disk form a good bimodal distribution. This bimodal distribution significantly reduces the garbage collection overhead.

Because WOLF works before initial writes go to disk, it can be integrated with other strategies smoothly to improve LFS performance. By reducing cleaning overhead, WOLF ameliorates the competition of disk bandwidth. Simulation experiments based on a wide range of real-world and synthetic workloads show that our strategy can reduce the overall write cost by 53% and improve write response time by 35.5%. The read performance is generally better than or comparable to the LFS. Our scheme still guarantees fast crash recovery, a key advantage of LFS.

We believe that our method can significantly improve the performance of those IO systems (such as some RAIDs) that use the LFS technology. It may also increase the chance of LFS success in the OS environments like Linux. Moreover, since logging is a commonly used technology to improve the I/O performance, we believe that our new scheme will have a broad impact on high performance I/O systems as well. We also plan to apply this technique to other general file systems like FFS in the future.

Acknowledgments

This work is supported in part by the National Science Foundation Career Award CCR-9984852, and an Ohio Board of Regents Computer Science Collaboration Grant. We would like to thank our shepherd Richard Golding for his many valuable suggestions on the final version of this paper. We would also like to thank the anonymous reviewers for their useful comments.

References

[1] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), Association for Computing Machinery SIGOPS, pp. 198–212.

[2] BLACKWELL, T., HARRIS, J., AND SELTZER, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA* (Berkeley, CA, USA, Jan. 1995), USENIX Association, Ed., USENIX, pp. 277–288.

[3] ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation* (Seattle, WA, Oct. 1996).

[4] GANGER, G. R., AND PATT, Y. N. Metadata update performance in file systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (Nov. 1994), pp. 49–60.

[5] GANGER, G. R., AND PATT, Y. N. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers 47*, 6 (June 1998), 667–678.

[6] GRIFFIOEN, J., AND APPLETON, R. The design, implementation, and evaluation of a predictive caching file system. Tech. Rep. CS-264-96, Department of Computer Science, University of Kentucky, June 1996.

[7] HARTMAN, J. H., AND OUSTERHOUT, J. K. letter to the editor. *Operating Systems Review 27*, 1 (1993), 7–9.

[8] HU, Y., AND YANG, Q. DCD—disk caching disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)* (Philadelphia, Pennsylvania, May 1996), pp. 169–178.

[9] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *Sixteenth ACM Symposium on Operating System Principles (SOSP '97)* (1997).

[10] MENON, J. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Aug. 1995), pp. 167–178.

[11] NIGHTINGALE, T., HU, Y., AND YANG, Q. The design and implementation of DCD device driver for UNIX. In *Proceedings of the 1999 USENIX Technical Conference* (Monterey, CA, Jan. 1999), pp. 295–308.

[12] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O bottleneck: A case for log-structured file systems. Tech. Rep. UCB/CSD 88/467, Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, Oct. 1988.

[13] R.LUMB, C., SCHINDLER, J., GANGER, G. R., AND NAGLE, D. F. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 2000 Conference on Operating System Design and Implementation (OSDI)* (San Diego, Oct. 2000).

[14] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Conference* (June 2000).

[15] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26 – 52.

[16] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proceedings of Winter 1993 USENIX* (San Diego, CA, Jan. 1993), pp. 307–326.

[17] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of 1995 USENIX* (New Orleans, LA, Jan. 1995), pp. 249–264.

[18] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC benchmark D standard specification, April 1995. Waterside Associates, Fremont, CA.

[19] WANG, J., AND HU, Y. Profs-performance-oriented data reorganization for log-structured file system on multizone disks. In *Proceedings of the 9th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS'2001)* (2001).

[20] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRaid hierarchical storage system. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 108–136.

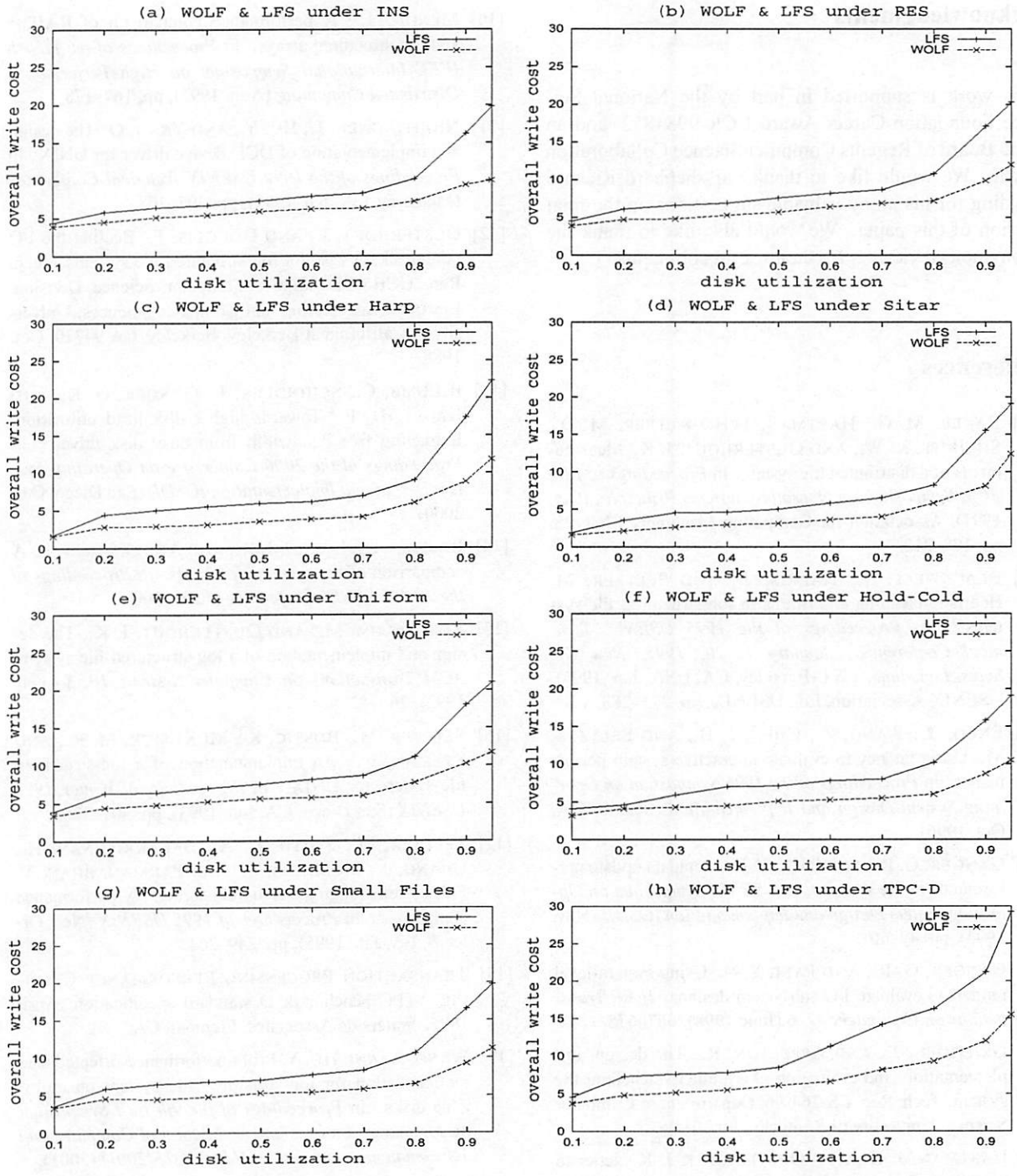


Figure 4: Overall Write-cost versus Disk Utilization under different workloads. *WOLF with 2 segment buffers.*

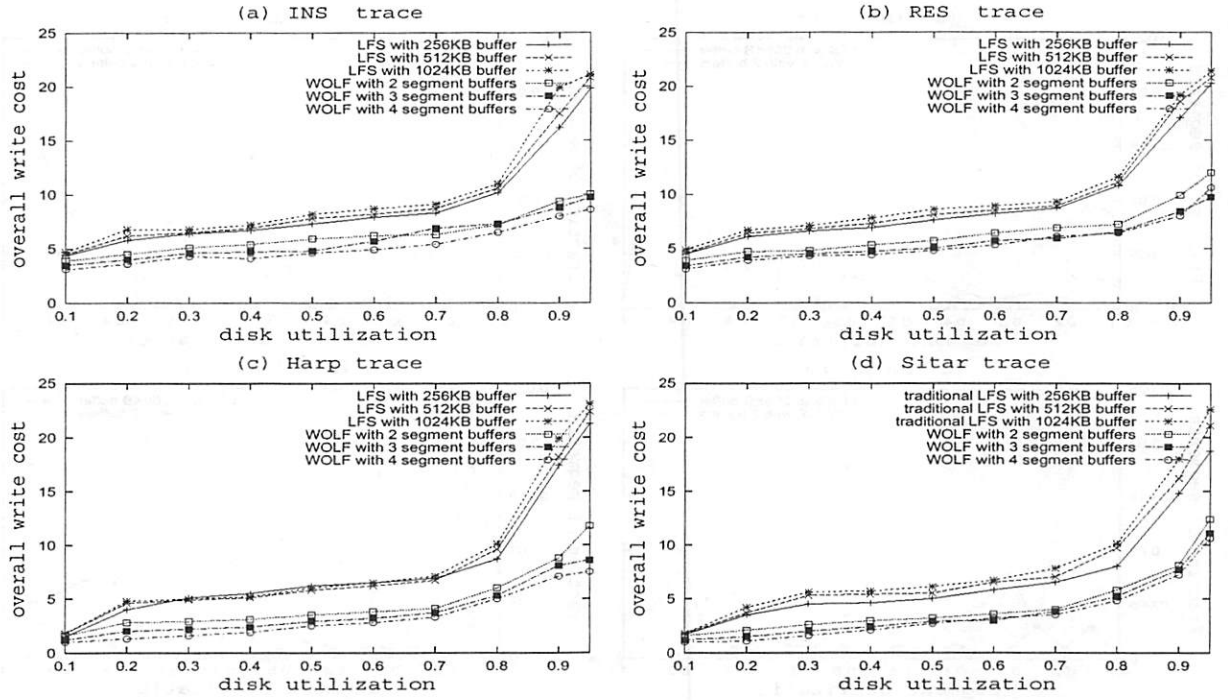


Figure 5: Overall write cost versus Disk Utilization.

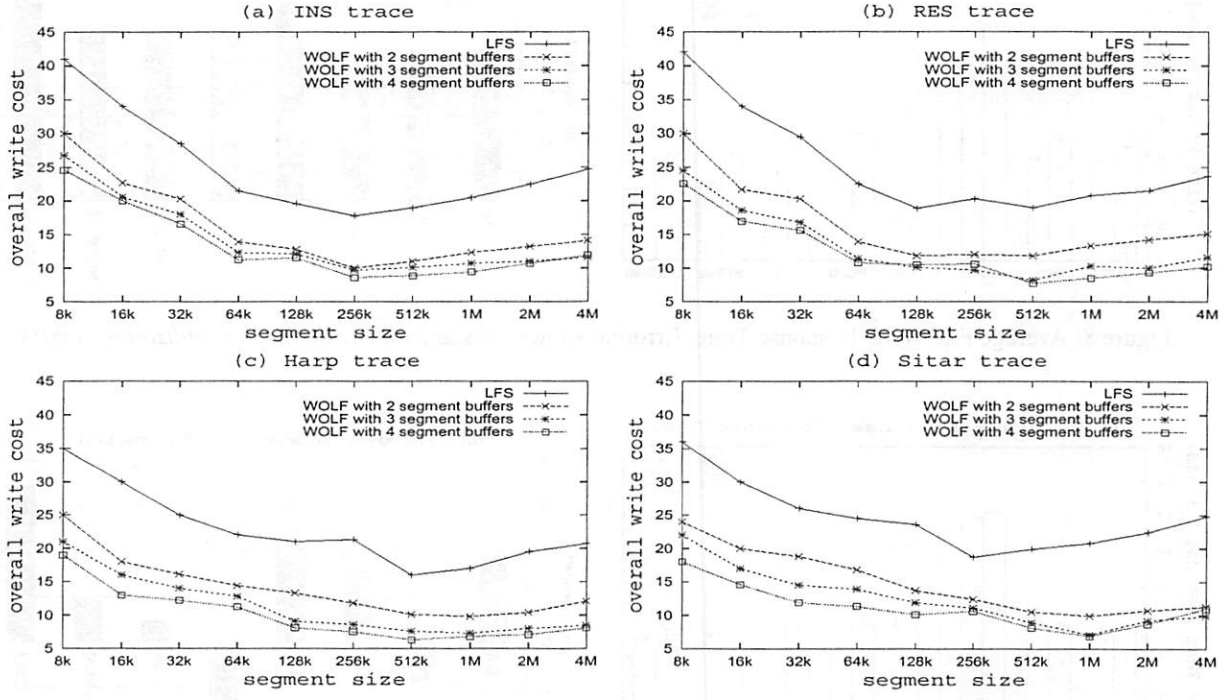


Figure 6: Overall write cost versus Segment Sizes. *Disk utilization is 95%.*

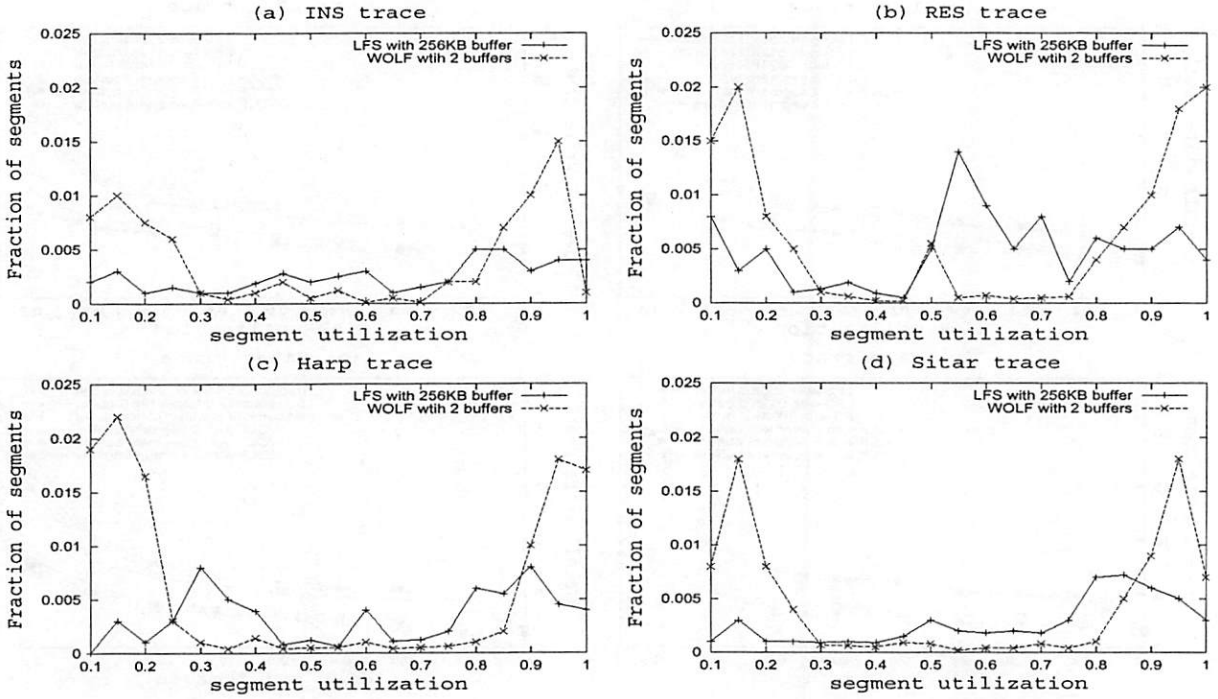


Figure 7: Segment Utilization versus Fraction of Segments. *Disk utilization is 80%.*

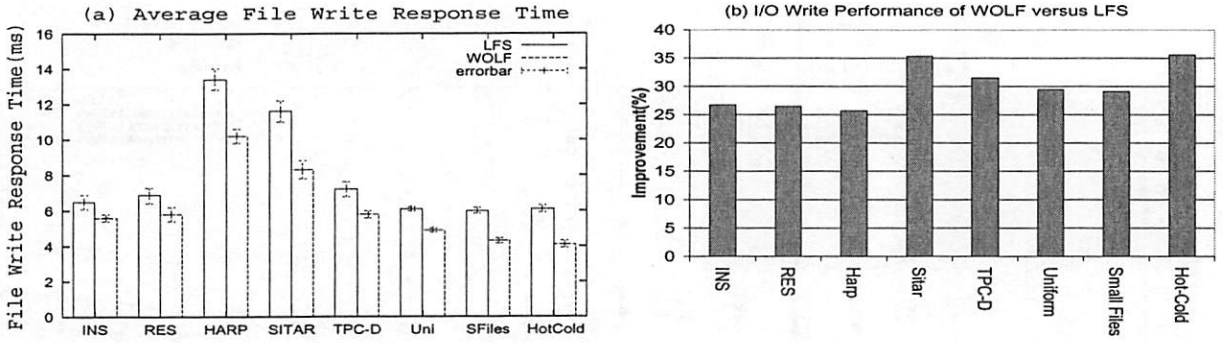


Figure 8: Average File Write Response Time. Errorbar shows the standard deviation. *Disk utilization is 90%.*

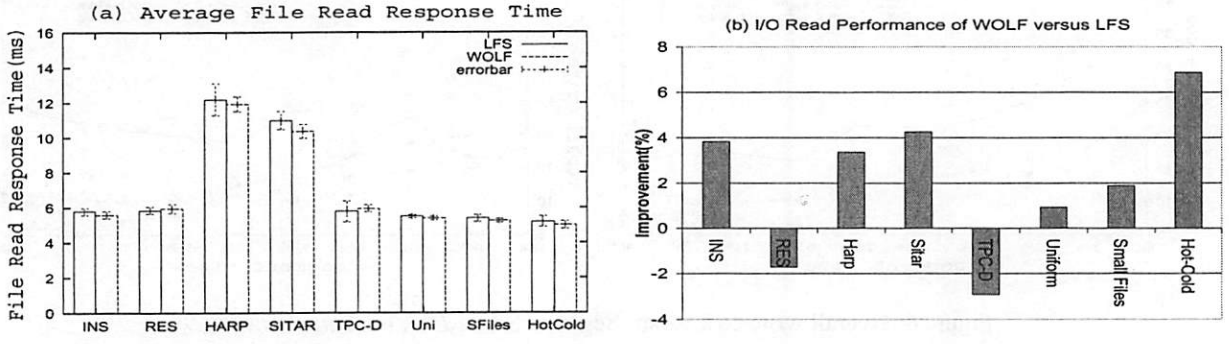


Figure 9: Average File Read Response Time. Errorbar shows the standard deviation. *Disk utilization is 90%.*

Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems

Brian C. Forney Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{bforney,dusseau,remzi}@cs.wisc.edu

Abstract

Modern storage environments are composed of a variety of devices with different performance characteristics. In this paper, we explore storage-aware caching algorithms, in which the file buffer replacement algorithm explicitly accounts for differences in performance across devices. We introduce a new family of storage-aware caching algorithms that partition the cache, with one partition per device. The algorithms set the partition sizes dynamically to balance work across the devices. Through simulation, we show that our storage-aware policies perform similarly to LANDLORD, a cost-aware algorithm previously shown to perform well in Web caching environments. We also demonstrate that partitions can be easily incorporated into the Clock replacement algorithm, thus increasing the likelihood of deploying cost-aware algorithms in modern operating systems.

1 Introduction

Modern computer systems interact with a broad and diverse set of storage devices, including local disks, remote file servers such as NFS [30] and AFS [16], archival storage on tapes, read-only media such as compact discs and DVDs, and even storage sites that are accessible across the Internet [4, 19, 25, 36]. As new storage components are introduced [11, 31, 34], their behaviors and properties will likely become even more divergent than they are today.

Although this set of devices is disparate, one commonality pervades them all: the time to access them is high, especially as compared to CPU cache and memory latencies [23]. Due to the cost of fetching blocks from storage media, *caching* of blocks in main memory often reduces execution time of individual applications and increases overall system performance – often by orders of magnitude.

However, while storage technology has dramatically changed over the past few decades, important aspects of the caching architectures used by modern operating systems have remained unchanged. Though there have been innovations in mechanism, including the integration of the file cache and virtual memory page cache [21], copy-on-write techniques [26], and software emulation of reference bits [7], there has been little change in policy, with most operating systems employing LRU or LRU-like algorithms to decide which block to replace.

The problem with LRU and related caching algorithms is that they are *cost-oblivious*: all blocks are treated as if they were fetched from identically performing devices and can be re-fetched with the same replacement cost as all other blocks. Unfortunately, this assumption is increasingly problematic, as the manifold device types described above have a correspondingly rich set of performance characteristics. As a simple example, consider a block fetched from a local disk as compared to one fetched from a remote, highly contended file server; in this case, the operating system should most likely prefer the block from the file server.

Within such heterogeneous environments, file systems require caching algorithms that are *aware* of the different replacement costs across file blocks. Given that the slowest device roughly determines the throughput of the system, storage-aware caching seeks to balance work across devices by adjusting the stream of block requests. Hence, in a heterogeneous environment, a storage-aware cache considers workload behavior *and* device characteristics to filter requests.

Thus, in this paper, we explore the integration of cost-aware algorithms into an operating system page cache using simulation. Our simulation accounts for real-world factors such as an integrated page cache and simplicity of design. We build on previous work in cost-aware caching from the web-cache and theory communities, demonstrating that a separate set of partitioned algorithms are as effective, yet simpler, than proposals in those research areas.

Our study is set in the context of a network-attached disk system. Network-attached disks are an increasingly important storage paradigm, and present clients with both static and dynamic forms of performance heterogeneity [5, 6, 13]. However, the algorithms we develop are general and can be applied across a broader range of storage devices.

Our main results are as follows. We show storage-aware caching is significantly more performance robust than cost-oblivious caching and as robust as a leading web-caching algorithm. Since operating systems have specific implementation needs, we develop and evaluate a version of storage-aware caching that extends the commonly implemented Clock algorithm.

The rest of this paper is organized as follows. In Section 2 we give an overview of the algorithms that we investigate in this paper. We then describe our algorithm for selecting partition sizes in Section 3. Section 4 describes the assumptions of our environment in more detail and explains our simulation framework. Simulation results are in Section 5. We compare and contrast our work to existing work in Section 6, and Section 7 contains future work. Finally, we conclude in Section 8.

2 Algorithm Overview

This section provides an overview of the algorithmic space we explore. First, we describe existing cost-aware algorithms as a basis of comparison. We then present our caching algorithms, which are based upon partitioning the cache according to replacement cost.

2.1 Existing Cost-Aware Algorithms

The theoretical community has studied cost-aware algorithms as *k-server problems* [12, 20]. A restricted class of k-server problems, *weighted caching*, is closely related to cost-aware caching. *LANDLORD* [40] is a significant algorithm in the literature, which we use for comparison. LANDLORD is closely related to the leading web caching algorithm [10].

LANDLORD combines replacement cost, cache object size, and locality by extending both LRU and FIFO to include cost and variable cache object sizes within a cache. Since we configured LANDLORD to use LRU, we describe the LRU version. LANDLORD associates a cost with each object, which is called L. When an object enters the cache, LANDLORD sets L to H, which is the retrieval cost of the object divided by the size of the object. If object eviction is needed, LANDLORD finds the object with the lowest L value, removes it, and ages all of the remaining objects. LANDLORD ages pages by decrementing the L value of all remaining objects by the L value of the evicted object. Upon reference of an object in the cache, LANDLORD restores its L value to H. LANDLORD degenerates to strict LRU when all H values are the same.

LANDLORD has attractive theoretical and experimental properties. As shown by theoretical analysis, when the size of cache objects are the same, LANDLORD is k-competitive, where k is the size of the cache. Thus, in a fixed object size cache, LANDLORD performs within a factor of k of the optimal off-line algorithm over all possible request sequences [39].

2.2 Overview of Aggregate Partitioned Algorithms

All of the cost-aware algorithms in the literature are place-anywhere. A *place-anywhere algorithm* has two characteristics: blocks may occupy any logical location in the cache, independent of their original source or cost, and costs are recorded on a page granularity. The advantage of place-anywhere algorithms is they calculate, in a single value, the trade-off between locality and cost. Thus, at replacement time, these algorithms bias eviction toward pages with low retrieval cost.

In contrast to a place-anywhere algorithm, an *aggregate partitioned algorithm* divides the cache into logical partitions, where blocks within a logical partition are from the same device and thus share the same replacement cost. The algorithm aggregates replacement cost since it is a function of a device's performance. An aggregate partitioned algorithm benefits from the aggregation of blocks and cost metadata in two ways: the amount of metadata is reduced and the value of the metadata more closely reflects the current replacement cost of a block from a device. Thus, the space overhead is proportional to the number of devices currently used and blocks are more likely to be replaced when the replacement cost is low.

Conversely, place-anywhere algorithms only record the cost when the page is brought into the cache. Thus, when the cache has a reasonably large number of pages, as is common today, a place-anywhere algorithm is more susceptible to inconsistent cost values. Aggregate partitioned algorithms avoid this problem by aggregating cost metadata on a per-device basis. As the performance of the device changes, the cost metadata is rarely inconsistent for more than a brief period of time. While a place-anywhere algorithm could recognize the change in cost for a device, and propagate the new cost to all pages in the cache, the cost update requires a significant number of pages to be updated, increasing overhead and implementation complexity.

Aggregate partitioned algorithms strive to set the relative size of each partition to balance work across devices. We define work as balanced when the cumulative delay for each device within a period of time is equal. To balance work, the size of each partition reflects the relative

cost of those blocks in a simple and efficient manner. For example, in a storage system with one slow disk and one fast disk, the cache is divided into two partitions, with the slow disk likely receiving a larger partition. We describe precisely how the relative sizes should be configured in the next section.

To choose a victim block, a storage-aware algorithm first selects a victim partition and then a victim block within that partition. The victim partition is chosen such that its resulting size relative to other partitions maintains the desired proportions. The individual victim within that partition can be selected with any replacement algorithm, such as LRU, LFU, or FIFO.

A few distinctions to prior work in virtual memory systems should be noted.

Unified and partitioned virtual memory systems: In the traditional sense, partitioned virtual memory systems distinguish between file system pages and virtual memory pages. The two are managed separately. Our storage-aware algorithms do not explicitly distinguish between file system pages and virtual memory pages. Rather, in order to balance work, our algorithms distinguish between pages based on which device supplied the page. Additionally, storage-aware caching algorithms change the size of partitions dynamically. Most partitioned virtual memory systems do not change the size of the file system cache and virtual memory partitions.

Local and global page replacement: Local page replacement at eviction time considers processes in isolation, while global page replacement applies replacement across processes. Our storage-aware algorithms make per-partition replacement decisions, which is similar to the traditional notion of local page replacement. However, the decisions are based on cost *and* locality, not solely on locality as in local page replacement schemes.

2.3 A Taxonomy of Aggregate Partitioned Algorithms

In our work, we investigate a taxonomy of aggregate partitioned algorithms and show that dynamic aggregate partitioning is needed. The taxonomy is described in this subsection.

Two basic approaches are possible for aggregate partitioning: *static* and *dynamic*. In a static scheme, the ratio across partitions is selected once according to a one-time notion of the costs. However, with no knowledge of the workload and its resulting miss rates for a given cache size, one cannot *a priori* determine the relative sizes that lead to balanced work. Thus, dynamic partitioning is needed, in which the ratio of partition sizes adjusts as the requests are monitored.

Dynamic partitioning have the following three benefits. First, dynamic partitioning can adjust to the dynamic performance variations, or faults, common in modern devices [6]. Second, dynamic partitioning can react to contention at devices due to hotspots in workloads. Finally, dynamic partitioning can compensate for the fact that the performance ratios across devices can change as a function of the access patterns.

Dynamic partitioning can be divided into *eager partitioning* and *lazy partitioning*. With eager partitioning, when new partition sizes are desired, the algorithm immediately reallocates pages using new cost information. An algorithm with a lazy partitioning scheme gradually reallocates pages on demand to the desired size in response to the workload. Eager partitioning simplifies choosing a victim partition, since it is the same as the location of the new page, at the cost of removing pages which may be useful. Conversely, a lazy partitioning scheme only removes pages from partitions when they are truly needed by another partition.

With lazy partitioning, a block may replace any other block, as long as blocks are replaced at the proper frequency to maintain the desired partition size ratios. Thus, on replacement, one must explicitly choose a victim partition. We investigate a strategy based on an *inverse lottery*, as previously proposed for resource allocation [33, 37]. The idea is that each partition is given a number of tickets in inverse proportion to its desired size. When a replacement is needed, a lottery is held by selecting a random ticket; the partition holding that ticket is picked as the victim. The victim then gives up its least valuable page, and the lazy partitioning algorithm allocates the page to a new logical partition.

3 Selecting Partition Sizes

The main challenge with partitioned approaches is in determining how the relative sizes of the partitions should be configured. Storage-aware caching can be viewed as performing selective filtering of requests to devices. Assuming the slowest device limits the system throughput, the goal of storage-aware caching is to set the partition sizes such that an equal amount of work is sent to each device. More formally, for each device, the number of cache misses multiplied by the average cost of each miss should be equal.

3.1 Algorithmic Details

Our basic approach uses a dynamic repartitioning algorithm. In the algorithm, the storage-aware cache observes the amount of work performed by each device over a fixed interval in the past and predicts how the relative sizes of the partitions should be adjusted so that the work is equal. The algorithm's work metric is cumulative delay over a period of time. The delay is related

to the total number of requests but includes request service time variation within a device and between devices. In this algorithm, streaming accesses that do not fit in the cache are problematic as the algorithm does not currently detect this type of access.

Our algorithm measures the time spent waiting for each device over the past W device requests, where W is the window size, and records this as the *wait time* per device. If all of the wait times are approximately equal, then the current partition ratios are deemed adequate and they remain the same. If the wait times are not the same, then the size of those partitions with relatively large wait times should be increased, and the size of those partitions with relatively small wait times should be decreased. Of course, the ratio of the wait times across all devices should be considered simultaneously.

Selecting the appropriate amount by which to increment and decrement each partition is a non-trivial search problem, given that one does not know how a given change in partition size affects the future miss rate, especially in the presence of dynamically changing workloads. Thus, with our initial approach, we employ the simplest algorithm that we have found to meet our needs. The challenge is to find an algorithm that adjusts partition sizes quickly enough to find the right proportions, but not so quickly that the algorithm overshoots those correct proportions.

To meet both of these goals simultaneously, our approach aggressively increases the size of a partition when the wait time for the corresponding device is increasing and otherwise reacts in a conservative manner. As such, our algorithm makes observations about the wait time for each device during an epoch, and an action is taken based on the observation. A new epoch begins after W device requests complete and between epochs the cache is repartitioned.

Repartitioning occurs in four steps. First, the algorithm computes the per-device wait time and the mean wait time across devices during the epoch. Second, the algorithm computes a *relative wait time* for each partition by dividing the per-device wait time by the mean wait time. Next, the algorithm determines which partitions are *page consumers* and how many pages to give each consumer. Page consumers are partitions which have a relative wait time above a threshold T. A threshold is used to filter normal variations in the wait time not due to changes in workload or device characteristics. If no page consumers are found, repartitioning stops and the new epoch begins. Finally, the algorithm finds partitions with below-average wait times, called *page suppliers*, and reallocates pages from them to consumers until the consumers reach their desired size.

While repartitioning the cache, the algorithm classifies each partition into one of four states, and may take corrective action to change the partition size. The four states are shown in Figure 1 and are described as follows.

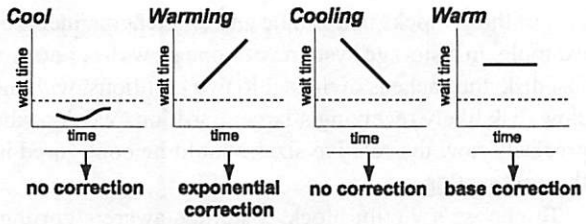


Figure 1: **Corrective actions taken by our repartitioning algorithm.** *This figure shows the four actions taken by our algorithm in response to four states. The graph shows the observation of the per-device wait time trend relative to the mean wait time as time progresses. A dotted line shows the mean wait time in each graph. Below the graphs are the actions taken in each state. While shown as fixed for overall clarity, the threshold is a constant value multiplied by the mean wait time.*

- **Cool: wait time below the threshold.** The wait time is within the normal operating regime. No corrective action is needed. Some cool partitions may become page suppliers, but none become page consumers.
- **Warming: wait time above threshold and increasing.** The algorithm infers an increasing wait time is due to changes in the workload or the device characteristics. Initially, the cache size is increased by I pages, where I is the *base correction amount*. If a partition continues to warm in subsequent epochs, the increase in cache size grows exponentially. A reclassification of a partition as warming from any other state restarts exponential correction.
- **Cooling: wait time above threshold and decreasing.** Corrective action during a set of epochs may have halted the increase in wait time for a partition and started a decline in wait time. The algorithm acts conservatively in the cooling state and does not change the partition size. A more aggressive approach that continues to increase the cache size for this state may over-correct and become unstable.
- **Warm: wait time above threshold and constant.** Based on experimental evidence, partitions are most often classified as cool, warming, or cooling. Thus, a constant wait time is unlikely to occur, and the partition moves to another state with a small change in the partition size. Thus, the algorithm increases the partition size by I.

The last step of the algorithm reallocates pages from page suppliers to page consumers. The algorithm biases collection of pages toward partitions that have the lowest relative wait times. To determine the number of pages removed from each page supplier, the algorithm first computes the *inverse relative wait time* ($IRWT$), which is just 1 - relative wait time, for each partition. Next, it sums the inverse relative wait times. Finally, the number of pages a partition j must supply is computed:

$$\frac{IRWT_j}{sum\ of\ IRWTs\ for\ suppliers} * \#\ of\ consumed\ pages$$

Note that there are three parameters to this algorithm: window size (W), threshold (T), and base increment amount (I). Each needs to be set with care. The value of W should be large to smooth out wait time variations and to sample a sufficient number of requests to determine accurately the effect of corrections. We have found $W = 1000$ provides sufficient smoothing and feedback. The value of I should be small since exponential correction is taken. We have found a value of 0.2% of the cache works well in practice. The threshold value, T, should be large enough to filter normal device performance fluctuation such as seek time. We have found $T = 5$ detects changes in wait time that warrant correction. Rather than use a fixed value of T, the algorithm could compute the threshold dynamically as the statistical variance of wait times and use the sum of the mean and the variance as the threshold. We do not discuss this adaptive approach here, but we plan to investigate it in the future.

3.2 Modifying Existing Replacement Algorithms

Not only do we desire to have a cost-aware cache that performs well, but also one that can be easily implemented in modern operating systems. Although attention has been paid to make it computationally efficient, LANDLORD needs a priority queue to efficiently find the lowest cost object and its use of L does not mesh well with common virtual memory hardware. Thus, it is not easy to combine LANDLORD with an existing code base.

Several modern operating systems, including Solaris [8], use a variant of the Clock page replacement policy in their unified page cache [7]. Thus, we desire an algorithm that can be incorporated easily into the Clock structure.

We introduce an extension of Clock that takes partitions into account, *Partitioned-Clock*. As in the base algorithm, Partitioned-Clock assumes that each page has a *use bit* which is set whenever the page is referenced; when a victim page is needed, the clock arm looks through successive pages for one that does not have its use bit set, clearing use bits as it sweeps. With Partitioned-Clock, each page also tracks the partition number to which it currently belongs; when a page is selected as a victim, not only must its use bit be cleared, but its partition number must match the partition number chosen for replacement (*e.g.*, as chosen by the lottery). We note that considering additional bits other than a single use bit is consistent with other variations of Clock that examine dirty bits or a history of multiple use bits.

There are a few optimizations that improve the performance of Partitioned-Clock. First, for the best approximation of lazy partitions, when searching for a replacement, only those pages belonging to the victim's partition should have their use bits cleared; clearing the use bits of all pages unnecessarily removes some of their usage history. Second, a separate clock hand for each partition also improves performance since it helps to further maintain the usage history of each partition.

As described previously, lazy partitions are simpler to implement than eager partitions when the ratio of their sizes is dynamic. Therefore, we focus on the Clock algorithm applied to lazy partitions. This version is termed *Lazy Clock*, and it uses inverse lottery scheduling to pick victims amongst partitions.

4 Evaluation Environment

This section describes our methodology for evaluating storage-aware caching. Specifically, it gives an overview of our simulator and describes our simulated storage environments.

4.1 Simulator

We have developed a trace-driven storage-system simulator to study the behavior of storage-aware caching. As configured, the simulated environment looks like a single client connected to sixteen network-attached storage devices. With our simulator, we are able to explore the performance impact of client workloads, data layout, caching algorithms, network characteristics, disk characteristics, and storage-system heterogeneity.

The simulator is driven by the workload of the client, which is specified in a trace file. The trace file represents data block requests that have been striped with RAID-0 across the full set of disks; each request specifies the starting offset and length of the data to read or write. To simulate a system under high demand, we consider a closed workload model, in which the completion of one disk request immediately triggers the next request.

The client has a local cache, with its replacement policies the focus of our investigation. We do not model the time for a cache hit, since it is small enough in a real system to be dwarfed by the cost of remote-block access. The time for a cache miss is the sum of network transit time plus the remote disk service time.

	Trace 1	Trace 2	Trace 3
Request distribution	uniform	exponential	exponential
Disk distribution	uniform	uniform	Gaussian
Locality	random	random	random
Mean request size	256 KB	34 KB	34 KB
Working set size	400 MB	425 MB	425 MB
# of requests	192,000	750,000	750,000

Table 1: **Characteristics of synthetic traces.** *This table summarizes the three synthetic traces used in the first set of experiments. The Gaussian distribution has a mean of disk 7 and a standard deviation of 3.*

Our storage device model roughly matches that described in Ruemmler and Wilkes [29]; we model cylinders and consider seek time, rotational delay, and bandwidth in calculating the transfer time for a given request. Specifically, if a disk request falls within the same cylinder as the previous request, we model it as sequential; *i.e.*, the seek and rotational delay are set to zero and the transfer time is determined by bandwidth. For non-sequential requests, the rotational delay is chosen uniformly at random from zero to a full rotation time; the seek time follows a non-linear model [29] and depends upon the cylinder distance between the current request and the previous request.

Our network model is based on LogGP [3] with endpoint contention. LogGP, which was designed to model communication within large parallel computers, depends on five parameters. L is the message latency through the network, o is the endpoint overhead, g is minimum time between message sends, G is the seconds per byte through the network, and P is the number of endpoints.

4.2 Workloads

To fully understand the impact of storage-aware caching algorithms, we study two sets of workloads: a variety of synthetic traces and a web server trace collected and analyzed by Roselli *et al.* [28]. We simply refer to the Roselli *et al.* web server trace as the Roselli trace.

We use synthetic workloads to control different request size distributions, working set size, locality distributions, and distribution of request across disks. The synthetic traces are summarized in Table 1. These traces are read only. Traces 2 and 3 have a variety of request sizes to stress small and large read requests, and Trace 3 adds a request imbalance across disks.

The Roselli trace is of an image server at the University of California, Berkeley from January 25, 1997. The image server ran a web server and a Postgres database, which stored the images. The trace alternates between large reads of several files, which are most likely the database tables, and small reads and writes.

Age (Years)	Bandwidth (MB/s)	Seek Avg (ms)	Rotation Avg (ms)
0	20.0	5.30	3.00
1	14.3	5.89	3.33
2	10.2	6.54	3.69
3	7.29	7.27	4.11
4	5.21	8.08	4.56
5	3.72	8.98	5.07
6	2.66	9.97	5.63
7	1.90	11.1	6.26
8	1.36	12.3	6.96
9	0.97	13.7	7.73
10	0.69	15.2	8.59

Table 2: **Aging an IBM 9LZX.** *We model the bandwidth, seek, and rotation time for a family of disks based on the IBM 9LZX manufactured in progressively older years. We assume bandwidth improves by a factor of 40% per year and seek and rotation time by 10% per year.*

4.3 Storage-System Characteristics

We have two goals in configuring the set of disks in our simulated environment. The first goal is to understand the full sensitivity of storage-aware caching algorithms to device heterogeneity; this requires a diverse range of configurations. The second goal is to understand how these algorithms perform in realistic scenarios; this requires a more focused set of tests.

To meet both of these goals simultaneously, we employ *device aging* and *performance-fault injection*. The idea behind device aging is to choose a base device (in our case, the IBM 9LZX) and age its performance over a range of years and use different collections of these disks to create configurations. The key, however, is that the performance of the base disk should not be scaled by a fixed amount; instead, each component (bandwidth, seek, and rotation time) should be scaled by its expected yearly improvement. Historical data suggests that a 40%/year improvement in bandwidth and roughly a 10%/year reduction in seek time and rotational latency is realistic (although perhaps on the aggressive side) [15]. Table 2 shows the performance characteristics of the aged devices used in our experiments. Note that although we consider progressively older disks (backwards aging), one could consider newer disks based upon the current year in a similar manner (forward aging).

Performance-fault injection allows us to dynamically

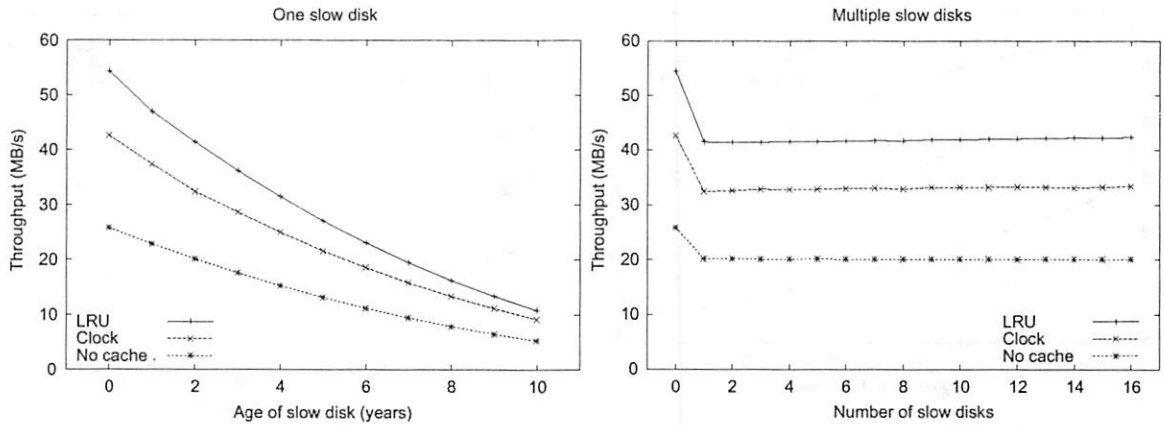


Figure 2: **Performance of LRU, Clock, and No Caching.** *The figures show the throughput of the storage system when Trace 1 is used. The figure on the left varies the age of a single disk along the x-axis. The figure on the right increases the number of two year old disks in the system.*

change the performance of a drive during an experiment. As described earlier, this could represent a disk stuttering before absolute failure, unexpected network traffic between the client and the drive, or a sudden workload imbalance.

4.4 Environment Configuration

This section describes the details of the simulator configuration. We configure the network so that it is not the bottleneck of the system and choose parameters that are similar to a 10 Gb/s Ethernet; thus, we set the bandwidth (*i.e.*, $1/G$) to 10 Gb/s, L to $1\mu s$, o to $0.4\mu s$, and g to 76 ns. In the future, we hope to investigate how network performance and caching interact in distributed storage systems.

We configured the simulator separately for the synthetic and Roselli traces. For the synthetic traces, we choose a sufficient number of requests to mitigate the effects of cold-start misses, and set the client cache size to 200 MB. For the Roselli trace, we set the cache size to 10 MB so that the hit rate is near 50%. If the hit rate is too high, few requests are sent to the disks, and thus, the heterogeneity of device performance is less of an issue. Since these traces did not include disk layout and file path information, we created a simple layout policy. The layout policy assumes RAID-0 striping. The policy lays out blocks in the order of first access.

We aged the disks in two scenarios. Both scenarios represent cases where the storage-system has been incrementally updated; that is, newer, faster devices have been added over time. In the first scenario, there is a single heterogeneous disk whose performance is aged across the entire range of years. In the second scenario, there are two groups of heterogeneous disks, one group with an age of zero, the other with an age of two years, and the relative size of the two groups is varied.

While these scenarios do not cover all real-world situations, they provide insight for common configurations. The first scenario mimics stuttering disks and increased workloads from other clients. The second scenario closely follows incremental upgrades of a disk array. Incremental upgrades often occur due to cost constraints that prohibit the replacement of an entire array when a small number of the disks fail.

5 Experiments

This section presents a progression of experiments demonstrating the effectiveness of storage-aware caching. We begin by motivating the need for cost-aware caching algorithms given heterogeneous devices. We then show that partitioned approaches can mask performance differences, but that configuring fixed partition ratios correctly is difficult even in a static environment. Next, we demonstrate that we can mask performance heterogeneity by adjusting the ratio of partition sizes according to on-line observations of the amount of work performed by each device. Finally, we show that partitioned approaches can be easily incorporated into operating system replacement policies and still perform quite well, and explore their performance robustness on a trace of a web server.

5.1 A Motivating Example

Our first set of experiments motivates the need for storage-aware caching algorithms given a storage system containing heterogeneous devices. Figure 2 shows the throughput obtained for Trace 2 using two common replacement policies that are not cost-aware, LRU and

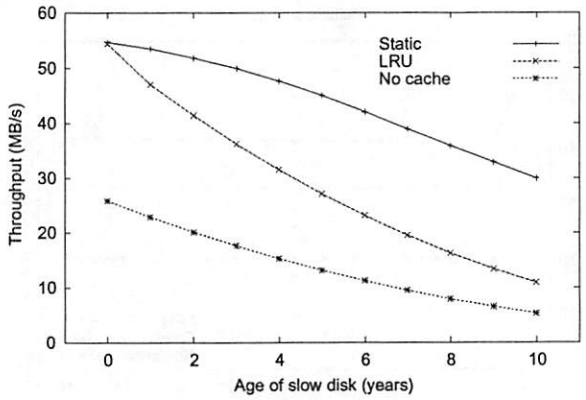


Figure 3: **Potential of partitioned approaches.** *One slow disk was aged as shown by the x-axis for three approaches: no cache, LRU caching, and a static partitioning of the cache according to disk performance. Trace 1 was used for this figure.*

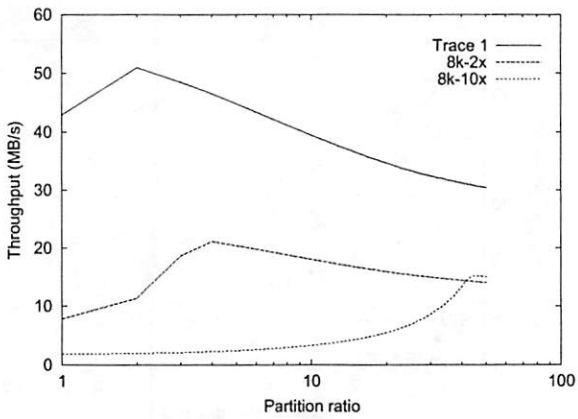


Figure 4: **Sensitivity of static partitioning on partition ratios and workload.** *The graph shows the performance of three different workloads run on 16 disks with one two-year old disk. The experiment varies the ratio of the slow disk partition to each of the fast disk partitions on the x-axis. The lower two lines both use 8 KB requests, but vary the ratio of requests sent to the slow disk versus the others, using either a ratio of 2:1 or 10:1.*

Clock, as well as with no caching. The graph on the left illustrates that as one of the disks is aged (and its seek, rotation time, and bandwidth decline) the throughput of the system drops dramatically, and the performance benefit of having a file cache decreases. For example, with LRU replacement, throughput drops from nearly 55 MB/s when all of the disks are equally fast down to only 11 MB/s when just a single disk has the performance of a 10-year old disk. Similarly, the graph on the right shows that the entire storage system runs at the rate of the slowest disk in the system; that is, the throughput with one slow disk and 15 fast disks is as poor as with all slow disks.

In contrast, a storage-aware caching algorithm should mask the performance of slow disks by allocating more of the cache to the slow disks; the slow disk thus has fewer requests to handle and does not harm the performance of the system as dramatically.

5.2 Configuring Partition Sizes

In our next set of experiments, we show that partitioned caching algorithms have the potential to mask heterogeneous performance, but that care must be taken in selecting the ratio of partition sizes. We begin by examining a static partitioning algorithm simply named *Static*.

In Figure 3, we show the performance of Static for Trace 1. In these experiments, the ratio of partition sizes is statically set and is directly proportional to the ratio of the expected service time of each disk. Since we use Trace 1, we know the mean request size is 256 KB. As such, we directly compute the expected transfer time from each disk as a function of its seek time, rotation delay, and peak bandwidth. The graph indicates a static partition strategy significantly improves performance relative to cost-oblivious algorithms such as LRU. However, Static performs well only when a priori mean request size and per-disk miss rate as a function of cache size is known. In the real world, this information is not known in advance.

The difficulty of correctly configuring a static partition strategy is illustrated in Figure 4. In this experiment, we examine a single storage configuration in which the one slow disk is two years older than the other disks. Along the x-axis, the graph varies the ratio of the partition sizes between the slow disk and each of the other disks in the system; for example, when this value is greater than one, the slow disk is given a correspondingly larger partition.

The three lines in the graph correspond to three different workloads, each of which has a different optimal value for the partition size ratios. The top line is the same workload as examined above; we verify that the highest throughput for this workload is approximately 50 MB/s, which matches that shown in Figure 3 with a two-year old disk. In the second and third lines, the distribution of requests across disks is changed such that the slow disk receives either twice or ten times as many requests as the other disks. The graph shows that each of these workloads has a different optimal partition ratio (*e.g.*, the best ratio for the top workload is 2:1 whereas the best ratio for the bottom workload is nearly 40:1). Further, the performance of each workload varies greatly with the partition ratio (*e.g.*, the performance of the 256 KB workload varies from 50 MB/s to 30 MB/s). This indicates we need

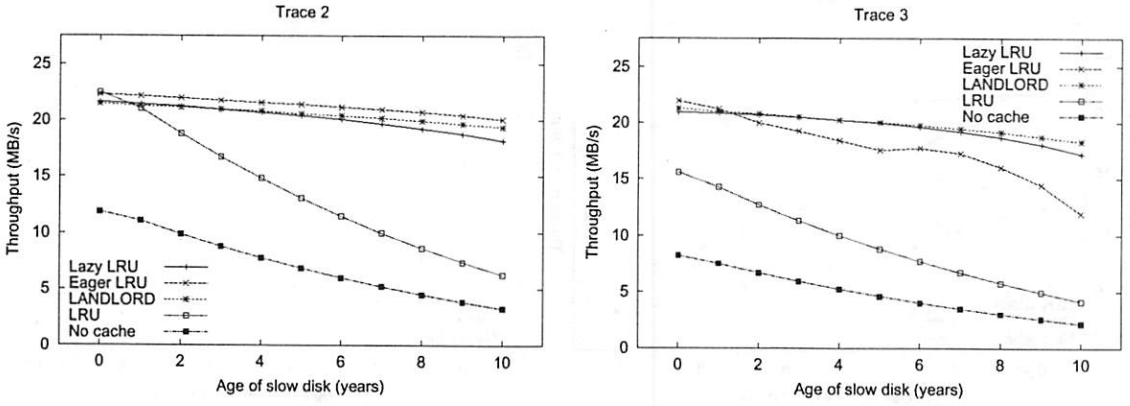


Figure 5: **Dynamic partitioning algorithms.** *The figure shows the performance of Eager LRU, Lazy LRU, and LANDLORD as one disk is aged. The left graph uses Trace 2 while the right graph uses Trace 3.*

an approach to select the partition ratios dynamically as a function of both the workload and the disk performance.

5.3 Partitioning to Balance Work

Our next set of experiments shows that by dynamically adjusting the size of each partition, our algorithms balance the amount of work performed by each disk and thus effectively hide heterogeneity. In doing so, we use the two classes of dynamic partitioning: eager partitioning and lazy partitioning. Lazy partitioning uses inverse lottery scheduling to pick victim partitions at replacement time. Both eager and lazy use LRU within their partitions. For simplicity, we refer to the first approach as *Eager LRU* and the second approach as *Lazy LRU*. In these experiments, we investigate Trace 2 and Trace 3 for a more realistic evaluation while continuing with well understood workload parameters.

Figure 5 compares the performance of the Eager LRU and Lazy LRU storage-aware algorithms to LRU and LANDLORD. In the left-most graph of the figure, we examine the workload with a uniform number of requests across disks. With this setup, the throughput with LRU degrades dramatically as the performance of the one slow disk is aged; specifically, throughput drops from approximately 23 MB/s to only 6 MB/s. Eager LRU and Lazy LRU are able to maintain the throughput of the system as the slow disk is aged; specifically, the performance of these algorithms is similar to that of LRU when all of the disks are the same speed, but with a ten-year-old disk, they are able to mask the impact of the slow disk and keep throughput between 16 and 20 MB/s.

The right graph of Figure 5 shows the challenges of a non-uniform number of requests across disks. Interestingly, even when all of the disks are identical, all of the cost-aware algorithms perform better than LRU. With this workload, the popular disks suffer from contention, and thus queueing delays make blocks from those disks more costly to fetch. By monitoring replacement cost, the cost-aware algorithms devote more of the cache to the popular disks and thus better balance the load across all of the disks. As with the previous workload, the performance benefits of cost-aware caching improve as one of the disks is aged. For a 10-year old disk, Eager LRU and LRU differ by over a factor of three.

Comparing the performance of Eager LRU, Lazy LRU, and LANDLORD, one sees that the performance of the three algorithms is similar, but not identical. The right graph most clearly shows the difference. Eager LRU is not as performance robust as Lazy LRU and LANDLORD. While Lazy LRU devotes the entire cache to the slow disk, Eager LRU continues to allocate a small amount of the cache to the fast disks. The immediate repartitioning of Eager LRU aggravates efforts to find a good partition size on Trace 3.

5.4 Clock-Based Replacement

As noted in Section 3.2, several operating systems use the Clock algorithm. However, Clock is not cost-aware. Thus, in this section, we evaluate the use of a lazy partitioned algorithm called Lazy Clock as a practical virtual memory page replacement algorithm. Again, we use Traces 2 and 3 and compare Lazy Clock to Clock. Experimental results are found in Figures 6 and 7.

In Figure 6, Lazy Clock performs well. As desired, Lazy Clock gives a greater proportion of the cache to slower devices and devices with more requests. Thus, Lazy Clock is able to mask performance differences even as the speed of the one slow disk degrades significantly. For example, with the imbalanced workload, Lazy Clock begins with a throughput of approximately 21 MB/s when all disks are identical and degrades to only about 17 MB/s when the slow disk is a full 10-years older

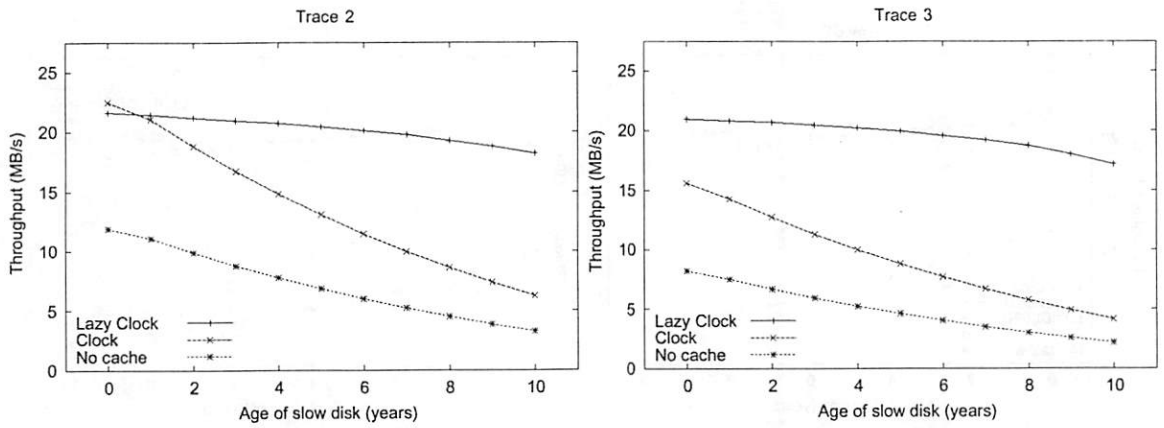


Figure 6: **Clock-based replacement algorithms.** *The figure shows the performance of Lazy Clock and Clock as one disk is aged. The same workloads are investigated as in Figure 5.*

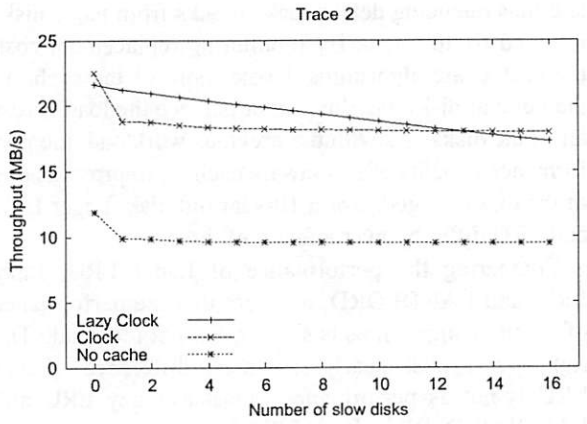


Figure 7: **Clock-based with multiple old disks.** *The figure shows the performance of Lazy Clock and Clock as the number of disks of age two years is increased. Trace 2 is used, which is the same workload as in the left graph of Figure 5.*

than the others. This throughput compares favorably to LANDLORD and Lazy LRU in Figure 5.

Figure 7 shows how Lazy Clock gracefully masks an increasing number of two-year old disks. While Clock is affected by any heterogeneity, the performance of Lazy Clock only slowly degrades. While the performance does not match Clock when the system is homogeneous, such as when there are zero or 16 two-year old disks, the performance is fairly close. Our experience has shown that a smaller base correction size when the devices are homogeneous can remove the discrepancy.

5.5 Dynamic Changes in Performance

To evaluate tolerance to performance faults, we show that our partitioned caching algorithms are able to react to changes in the relative performance of storage devices, in some cases more effectively than LANDLORD is able. In these experiments, we begin with a cluster of homogeneous disks and Trace 1. We inject a performance fault on one of the disks (disk 6) at a simulated time of 500 seconds (approximately half way through the simulation). The performance fault has the effect of slowing down that disk by a factor of two.

In the first graph of Figure 8 we show how Eager LRU adjusts the partition ratios for this change in performance. As one can see, the 16 partitions are all initially equal. After the performance fault and a window of W = 1000 disk requests have passed for observation, the algorithm observes that the waiting time of disk 6 is significantly higher than the average waiting time. The partition for disk 6 is then increased by a small amount and the partitions for all of the other disks are decreased by the necessary number of cache entries. The algorithm continues measuring the wait time of each disk and increasing the partition size of disk 6 until all of the wait times are approximately equal. The time-line shows that the correct partition ratio is found quickly.

In the second graph of Figure 8 we summarize our results by comparing the performance of LANDLORD, Eager LRU, Clock, and Lazy Clock. We plot throughput for the same workload as above when a performance fault occurs on one disk, four disks, or eight disks simultaneously. When the number of affected disks is small relative to the total number of disks in the system, aggregating replacement-cost information is beneficial. Specifically, Eager LRU achieves a throughput of approximately 54 MB/s with one performance fault, while LANDLORD maintains only 48 MB/s. Lazy Clock performs nearly as well as the others, and finally the cost-oblivious Clock algorithm performs the least well.

When the number of faulty disks increases to eight,

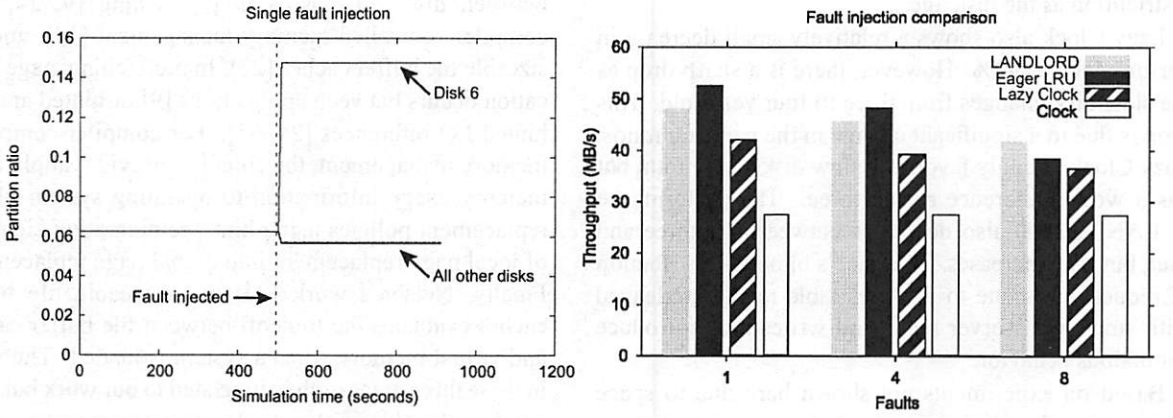


Figure 8: **Performance with dynamic faults.** *We consider a cluster of initially homogeneous disks and a performance fault at time 500 seconds that slows a disk by a factor of two. In the graph on the left, the performance fault occurs on a single disk and we show the partition sizes chosen by the Eager LRU algorithm. In the graph on the right, the performance fault occurs on either one, two, or four disks. In all cases, Trace 1 was used.*

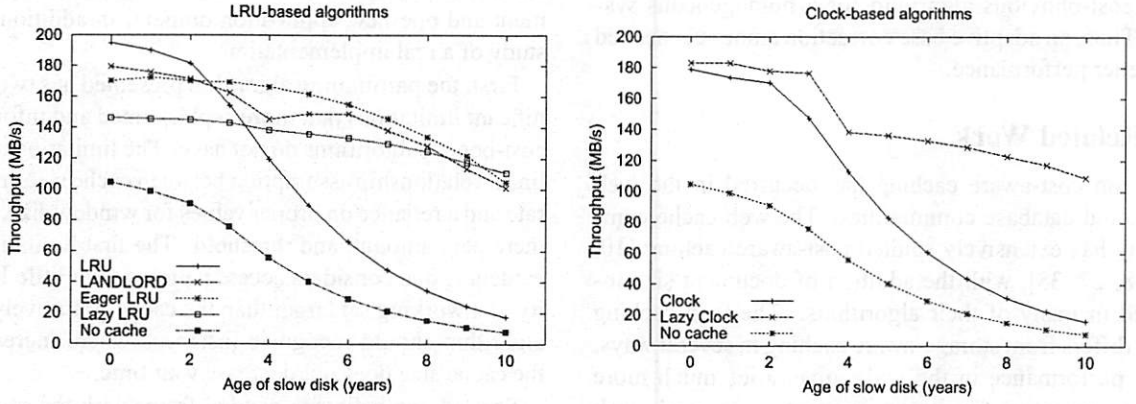


Figure 9: **Web server workload.** *This figure shows the performance of LRU- and Clock-based algorithms when run on a file system trace of a web server providing art images. The results of LRU-based algorithms are in the graph on the left side. Clock-based algorithms are in the right graph.*

the results change. Eager LRU does not mask the performance faults as well as LANDLORD. When there are two groups of disks with similar performance characteristics, our correction algorithm does not detect the severity of heterogeneity, making smaller corrections than are needed. An adaptive threshold value may help.

5.6 Real-World Performance

We conclude our experiments with an examination of our partitioned algorithms on a web workload. As the web server received a modest number of requests, this trace is shorter than our synthetic traces. Our partitioned algorithms are partially penalized by the shortness of the trace as they first need to move the partition sizes from an initial state. Similar to other experiments, we only investigate the aging of a single disk.

The results are shown in Figure 9. As expected, the cost-oblivious algorithms show a sharp drop off in performance as the age of the slow disk increases. LRU's performance falls to 10% of its peak performance over the age range and, for a more realistic range, falls nearly 30% when aged from zero to four years old. Clock shows a similar, though slightly less dramatic decline.

As expected, the adaptive algorithms show more robust performance. The performance degrades by 15% for Lazy LRU, 38% for Eager LRU, and 45% for LANDLORD. However, Lazy LRU performs more poorly than Eager LRU or LANDLORD. Lazy LRU has a poor interaction with the repartitioning algorithm and devotes the entire cache to the slow disk even when the disks are homogeneous. Eager LRU, however, distributes pages more evenly between pages and gradually changes the

distribution as the disk ages.

Lazy Clock also shows a relatively small decrease in performance of 40%. However, there is a sharp drop as the slow disk changes from three to four years old. This drop is due to a significant change in the partition ratios. Lazy Clock strongly favors the slow disk at age four, but has a weak preference at age three. The performance of LANDLORD also decreases between age three and four, but then increases. The trace's bimodal distribution of request sizes due to Postgres table reads interleaved with small web server reads and writes may introduce anomalous behavior.

Based on experiments not shown here due to space constraints, the performance of dynamic partitioning algorithms is sensitive to the base correction amount. The experiment in Figure 9 uses a fixed base correction amount across the range of slow disk ages. If the base correction amount starts small and increases as the disk ages, performance improves and nearly matches the base cost-oblivious algorithm for a homogeneous system. Thus, an adaptive base correction amount is needed for better performance.

6 Related Work

Work on cost-aware caching has occurred in the web cache and database communities. The web cache community has extensively studied cost-aware caching [10, 17, 18, 27, 38], with the addition of document size included in many of their algorithms. The web caching work differs from storage-aware caching in several ways. First, performance in the wide area varies much more than is common for storage systems. Second, web caching often uses whole document caching, which differs from fixed-size blocks used in storage systems. Finally, in web caching, the replacement cost of one web page is not strongly correlated with the replacement costs of other pages.

Broadcast disks [1, 2] continuously deliver data to clients through an asymmetric link following a broadcast schedule that is best able to meet the client's needs. When a client's needs are not met by the broadcast schedule, the client cache strives to manage the cache contents to mask the non-ideal broadcast schedule. Using knowledge of the broadcast schedule and probability of access, the cache manages its contents using an algorithm that generalizes LRU. Storage-aware caching differs in two ways. First, while it partitions cache pages by device, broadcast disks use a page rather than a device granularity to track replacement costs. Second, broadcast disks assume an infrequently changing broadcast schedule, whereas storage-aware caching must react to frequent changes in workload and device performance.

Recently, researchers have studied allocation of pages between different classes in prefetching [9, 24, 35], compiler-controlled memory management [14], and resizeable file buffer caches [22]. In prefetching, page allocation occurs between applications [9] or hinted and unhinted I/O references [24, 35]. For compiler-controlled memory management, the compiler provides application memory usage information to operating system global replacement policies using hints, reintegrating elements of local page replacement into global page replacement. Finally, Nelson's work [22] on resizeable file buffer caches evaluates the tradeoff between file buffer caches and virtual memory when a system is loaded. The work in these three areas is closely related to our work but does not directly address storage device heterogeneity.

7 Future Work

While storage-aware caching increases the storage system performance robustness by adapting to performance differences of devices, we see further areas of improvement and one new application domain, in addition to a study of a real implementation.

First, the partitioning algorithm presented has two significant limitations that more sophisticated and informed cost-benefit algorithms do not have. The limitations are a linear relationship assumption between cache size and hit rate and a reliance on proper values for window size, base increment amount, and threshold. The first limitation is evident if one considers access patterns with little locality or a working set larger than the cache. Intuitively, the algorithm should recognize instances where increasing the cache size does not decrease wait time.

Second, we believe a general framework for storage-aware caching where existing cost-oblivious policies can manage individual partitions should be studied. A framework approach has modularity as its primary strength; existing non-cache aware policies such as LRU, Clock, MRU, and EELRU [32] can be used with minimal effort and changes.

Third, our work has concentrated on non-cooperative client caching. We believe the combination of cooperative caching and cost-aware caching will lead to better performance robustness, especially for disk arrays where individual cache sizes are small.

Fourth, storage-aware caching can be applied to low-power environments. Storage-aware caching can be extended to include power as a retrieval cost. Thus, devices that are higher in power will likely be accessed less frequently and stay in low-power mode longer and more frequently.

Finally, since caching algorithms are affected by prefetching and layout decisions, we would like to explore the advantages and tradeoffs of integrated prefetching, layout, and caching decisions, in light of device het-

erogeneity. Previous caching and prefetching work [9, 24] in homogeneous environments has shown the benefits of integration. We believe this benefit extends to heterogenous environments.

8 Conclusions

Given the diverse characteristics of modern storage devices, we believe the time is ripe to re-investigate caching algorithms. To optimize performance, the task of a cost-aware cache is to control which blocks are cached, such that the amount of work performed by each storage device is roughly equal. In this paper, we have presented a family of cost-aware caching algorithms that are based on the notion of explicitly partitioning the cache; the size of each partition is configured such that it directly corresponds to the relative cost and usefulness of the data in that partition. These approaches have two advantages. First, partitions are able to aggregate replacement-cost information across many entries in the cache, reducing the amount of information that must be tracked and allowing the most recent cost information to be used for all blocks from the same device. Second, and most important, a virtual partition approach can be easily implemented within the Clock replacement policy, increasing the likelihood of adoption in real systems.

Acknowledgements

We thank members of the Wisconsin Network Disks research group for helpful discussions, contributions, and paper comments. In particular, Florentina I. Popovici provided the IBM 9LZX disk profile information used in our simulations, and Leslie Cheung wrote the layout code used in the Berkeley trace experiments. Omer Zaki contributed to early versions of this work. Comments from anonymous reviewers and particularly the tireless shepherding of Jeff Chase greatly improved the quality of the paper. The Condor distributed execution system was used to run simulations. Members of the Condor project, especially Todd Tannenbaum, supported our use of Condor. This work is sponsored by NSF CCR-0092840, CCR-0098274, ITR-0086044, and the Wisconsin Alumni Research Foundation.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. B. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, May 22-25, 1995*, pages 199–210. ACM Press, 1995.

[2] S. Acharya, M. J. Franklin, and S. B. Zdonik. Dissemination-based Data Delivery Using Broadcast Disks. *IEEE Personal Communications*, 2(6), December 1995.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation. In *Papers of the 7th Annual ACM symposium on Parallel Algorithms and Architectures, Santa Barbara, CA, June 24 - 26, 1995*, pages 95–105. ACM Press, 1995.

[4] Apple Computer Corporation. iDisk. http://itools.mac.com.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, J. Bent, B. Forney, S. Muthukrishnan, F. Popovici, and O. Zaki. Manageable Storage via Adaptation in WiND. In *Proceedings of IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid' 2001)*, pages 169–177, May 15-18 2001.

[6] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *Workshop on Hot Topics in Operating Systems (HotOS 8)*, Schloss Elmau, Germany, May 2001.

[7] O. Babaoglu and W. Joy. Converting a Swap-Based System to do Paging in an Architecture lacking Page-Referenced Bits. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 78–86, Pacific Grove, CA, December 1981.

[8] J. L. Bertoni. Understanding Solaris Filesystems and Paging. Technical Report TR-98-55, Sun Microsystems, 1998.

[9] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[10] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 193–206, Berkeley, CA, USA, 1997.

[11] L. R. Carley, G. R. Ganger, and D. F. Nagle. MEMS-based Integrated-Circuit Mass-Storage Systems. *Communications of the ACM*, 43(11):72–80, November 2000.

[12] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New Results on Server Problems. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 291–300, San Francisco, CA, USA, Jan. 1990.

[13] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Usenix 2001 Annual Technical Conference*, June 2001.

[14] A. Demke Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, Berkeley, CA, October 23–25 2000.

[15] E. Grochowski. IBM Leadership in Disk Stroage Technology. IBM Corporation, 2000.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.

[17] S. Jin and A. Bestavros. Popularity-Aware GreedyDual-Size Algorithms for Web Access. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[18] T. Kelly, Y. M. Chan, S. Jamin, and J. K. MacKie-Mason. Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value. In *Fourth International Web Caching Workshop, San Diego, California, 31 March - 2 April 1999*, 1999.

[19] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[20] M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for On-line Problems. In *Proceedings of the twentieth annual ACM Symposium on Theory of Computing, Chicago, Illinois, May 2–4, 1988*, pages 322–333, New York, NY, USA, 1988.

[21] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[22] M. N. Nelson. Virtual Memory vs. The File System. Technical Report 90.4, Compaq Computer Corporation, 2000.

[23] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *1994 ACM SIGMOD Conference*, May 1994.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[25] Pro Softnet Corporation. iBackup. http://www.ibackup.com/.

[26] R. Rashid, A. Tevanian, M. Young, D. Golub, D. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 31–39, Palo Alto, CA, October 1987. Association for Computing Machinery, IEEE.

[27] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[28] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 41–54, Berkeley, CA, June 18–23 2000.

[29] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[30] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Berkeley, CA, USA, June 1985. USENIX Association.

[31] J. L. Sanford, P. F. Greier, K. H. Yang, M. Lu, R. S. Olyha, Jr., C. Narayan, J. A. Hoffnagle, P. M. Alt, and R. L. Melcher. A One-megapixel Reflective Spatial Light Modulator System For Holographic Storage. *IBM Journal of Research and Development*, 42(3/4):411–426, 1998.

[32] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 122–133, New York, May 1–4 1999. ACM Press.

[33] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *Proceedings of the 2000 USENIX Annual Technical Conference (San Diego, California), pp. 337-350*, 2000.

[34] The Infiniband Trade Association. http://www.infinibandta.org, June 2001.

[35] A. Tomkins, R. H. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114. ACM Press, June 1997.

[36] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, July 1998.

[37] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Nov 1994.

[38] R. P. Wooster and M. Abrams. Proxy Caching that Estimates Page Load Delays. In *Proceedings of the 6th International WWW Conference*, April 1997.

[39] N. E. Young. The K-server Dual and Loose Competitiveness for Paging. *Algorithmica*, 11(6):525–541, June 1994.

[40] N. E. Young. On-line File Caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, January 17-19, 1999*, pages 82–86. ACM Press, 1999.

Timing-accurate Storage Emulation

John Linwood Griffin, Jiri Schindler, Steven W. Schlosser,
John S. Bucy, Gregory R. Ganger
Carnegie Mellon University

Abstract

Timing-accurate storage emulation fills an important gap in the set of common performance evaluation techniques for proposed storage designs: it allows a researcher to experiment with not-yet-existing storage components in the context of real systems executing real applications. As its name suggests, a timing-accurate storage emulator appears to the system to be a real storage component with service times matching a simulation model of that component. This paper promotes timing-accurate storage emulation by describing its unique features, demonstrating its feasibility, and illustrating its value. A prototype, called the Memulator, is described and shown to produce service times within 2% of those computed by its component simulator for over 99% of requests. Two sets of measurements enabled by the Memulator illustrate its power: (1) application performance on a modern Linux system equipped with a MEMS-based storage device (no such device exists at this time), and (2) application performance on a modern Linux system equipped with a disk whose firmware has been modified (we have no access to firmware source code).

1 Introduction

Despite decades of practice, performance evaluation of proposed storage subsystems is almost always incomplete and disconnected from reality. In particular, future storage technologies and potential firmware extensions usually cannot be prototyped by researchers, so any evaluation must rely upon simulation or analytic models of the prospective subsystem. Unfortunately, this reliance commonly limits consideration of real application workloads and complex "real system" effects, both of which can hide or undo benefits predicted by simulating storage components in isolation. For this reason, such localized evaluation has long been considered unacceptable in other disciplines, such as networking, architecture, and even file systems.

Timing-accurate storage emulation offers a solution to this dilemma, allowing simulated storage components to be plugged into real systems, which can then be used for complete, application-based experiments. As illustrated in Figure 1, a *storage emulator* transparently fills the role of a real storage component (e.g., a SCSI disk), correctly mimicking the interface and retaining stored data to respond to future reads. A *timing-accurate* storage emulator responds to each request after its simulator-computed service time passes; the performance observed by the system should match the simulation model. To accomplish this, the emulator must synchronize the simulator's internal time with the real-world clock, inserting requests into the simulator when they arrive and reporting completions when the simulator determines they are done. If the simulator's model represents a real component, the system-observed performance will be of that component. Thus, the results from application benchmarking will represent the end-to-end performance effect of using that component in a real system.

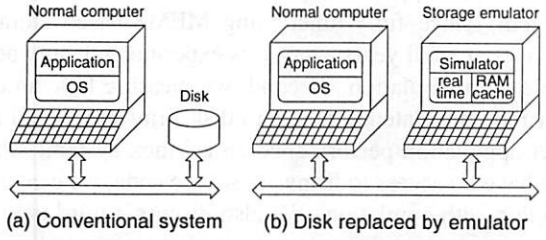


Figure 1: ***A system with (a) real storage or (b) emulated storage.*** *The emulator transparently replaces storage devices in a real system. By reporting request completions at the correct times, the performance of different devices can be mimicked, enabling full system-level evaluations of proposed storage subsystem modifications.*

This paper makes a case for timing-accurate storage emulation and demonstrates that it works in practice. It describes general design issues and details the implementation of our prototype emulator. Our original goal was thorough evaluation of operating system algorithms for not-yet-existing MEMS-based storage devices [11, 12]—this led to the prototype's name: *Memulator*. The Memulator integrates the DiskSim simulator [10], a real-time timing loop, and a large RAM cache to achieve flexible, timing-accurate storage emulation. It can emulate any storage component that DiskSim can simulate, including MEMS-based storage, disk arrays, and many

modern disk drives. Calibration measurements indicate that the Memulator's response times are within 2% of the DiskSim times for over 99% of requests. Using DiskSim's validated disk models, we also verify that system performance is the same with the Memulator as with a real storage device.

We illustrate the power of timing-accurate storage emulation with two experiments that the Memulator makes possible. First, we measure how MEMS-based storage would affect application performance on a current Linux system; since fully-functioning MEMS-based storage devices are still years away, this experiment is only possible with emulation. Second, we measure how an extension (zero-latency reads) to disk firmware would affect application performance on a Linux system; since we have no access to firmware source code, we can only do this with emulation. We also discuss a third type of experiment, interface extensions, that requires changes to both the host OS and the storage subsystem; without emulation (or complete implementation), thorough evaluation of interface extensions is not possible.

The remainder of this paper is organized as follows. Section 2 makes a case for timing-accurate storage emulation. Section 3 discusses the design of timing-accurate storage emulators in general. Section 4 describes the Memulator in detail. Section 5 validates the response times of the Memulator relative to simulated device performance. Section 6 describes experiments enabled by the Memulator. Section 7 summarizes this paper's contributions.

2 A case for emulation

Storage emulation is rarely used for performance evaluation of prospective storage system designs. This section makes a case for more frequent use, arguing that timing-accurate storage emulation offers a unique performance evaluation capability: experimentation with as-yet-unavailable storage components in the context of real systems. Such experimentation is important because complex system characteristics can hide or reduce predicted benefits of new storage components [9]. Further, some new storage architectures and interfaces require both OS modifications and new (or modified) storage components—until the new components are available, only emulation allows such collaborative advances to be tested and their performance evaluated.

2.1 Storage performance evaluation

Figure 2 illustrates a spectrum of techniques for evaluating storage designs, ranging from quick-and-dirty estimates to real application measurements on a complete system. Techniques to the left generally demand less of the evaluator: less effort to set up and employ, less time to produce a result, and less need for the evaluated storage system to be feasible. Techniques to the right generally produce more believable results: more accurate, more inclusive of complex system effects, and more representative of the effects under real workloads.

The six techniques shown are each appropriate in some circumstances, as each offers a different mixture of these features. For example, storage simulation allows hypothetical storage systems to be evaluated quickly and efficiently. Even futuristic technologies and modifications to proprietary firmware can be explored. Simulation results, however, must be taken with a grain of salt, since the simulation may abstract away important characteristics of the storage components, overall system, or workload. In particular, representative workloads are rarely used, since synthetic workload generation is still an open problem, I/O traces ignore system feedback effects [9], and available traces are often out-of-date—in fact, many storage researchers still rely on the decade-old "HP traces" from 1992 [21]. As a different example, experimenting with prototypes allows one to evaluate designs in the context of full systems and real workloads. Doing so, of course, requires considerable investment in prototype development and experiment configuration.

As indicated in Figure 2, storage emulation offers an interesting mix of features: the flexibility of simulation and the reality of experimental measurements. That is, storage emulation allows futuristic storage designs to be evaluated in the context of real OSes and applications. This enables two types of experiments. First, end-to-end measurements can be made of the effects of non-existent storage components in existing systems. Such components are usually simulated in isolation and evaluated under non-representative workloads. Second, end-to-end measurements can be made of the effects of non-existent storage components in modified systems. For example, storage interface changes often require that both the storage components and the OS be modified to utilize the new interface. Experimentation is impossible without the ability to modify both components, which is a very real problem with the proprietary firmware of most disks and disk array controllers. Section 6 explores concrete examples of both types of experiments.

We are aware of only one other technique offering a similar mix of features: complete machine simulation [3, 17, 19]. Under this technique, the hardware of a computer system is simulated in enough detail to boot a real OS and run applications. If the simulation progresses according to timing-accurate models of the key system components (e.g., CPUs, caches, buses, memory system, I/O interconnects, I/O components), it can be used for performance evaluation. Because it boots

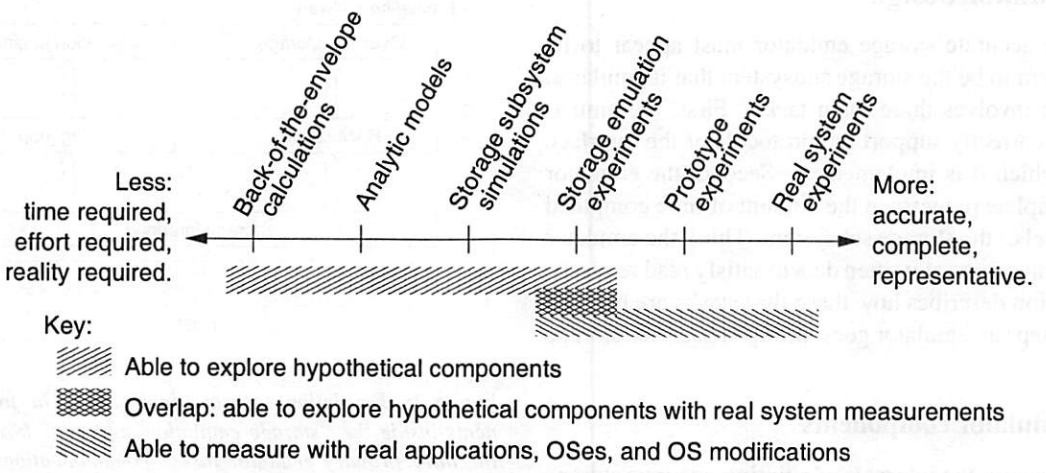


Figure 2: ***Storage performance evaluation techniques.*** *This illustration linearizes techniques across a spectrum from the ⟨quickest,easiest,most flexible⟩ to the most ⟨accurate,complete,representative⟩. In this spectrum, storage emulation provides the unique ability to explore nonexistent storage components in the context of full systems executing real applications.*

a real OS and runs real applications, a complete machine simulator enables the same types of experiments as storage emulation. Further, by manipulating simulator parameters, the effects of new storage devices on hypothetical machines (e.g., with 10 GHz CPUs) can be evaluated [20, 22]. Unfortunately, substantial effort is required to build and maintain a complete machine simulator, both in terms of correctly executing programs and correctly accounting for time. For example, the SimOS machine simulator required extensive effort to create and validate; just a few years later, its hardware models are out of date, the CPU instruction set it emulates is being phased out, and source code for the OS that it boots is difficult to acquire. In addition, these simulators usually run more slowly than real systems, increasing evaluation time. Storage emulation does not share these difficulties.

2.2 Related emulation

In a sense, storage emulation is commonplace. For example, the standard SCSI interface allowed disk arrays to rapidly enter the storage market by supporting a disk-like interface to systems. Similarly, the NFS remote procedure call (RPC) interface allowed dedicated filer appliances [13] to look like traditional NFS file servers. In addition, we have been told anecdotal stories of emulation's use in industry for development and correctness testing of new product designs. However, these examples represent only the "storage emulation" half of timing-accurate storage emulation.

The "timing-accurate" half has been much utilized by networking researchers [1, 6, 18]. Timing-accurate network emulation parallels our description of timing-accurate storage emulation: real hosts interconnected by the emulated network observe normal packet send/receive semantics and performance that accurately reflects a simulation model. The observable performance effects include propagation delays, bandwidths, and packet losses. Like timing-accurate storage emulation, timing-accurate network emulation enables real system benchmarking that would not otherwise be possible—in particular, deploying a substantial network just for experiments is simply not feasible.

We are aware of only a few previous cases of timing-accurate storage emulation being used for performance evaluation. The most relevant example is the evaluation of eager writing by Wang et al. [25]. Under eager writing, data is written to a disk location that is close to the disk head's current location. To evaluate the benefits of having disk firmware support for eager writing, Wang et al. embedded a disk simulator in Solaris 2.6, augmented it with a RAM disk, and arranged (using the `sleep()` system call) to have completions reported after delays computed by the simulator. Although some details differ, this is similar to the Memulator's design. A less direct example is the common practice of emulating non-volatile RAM by simply pretending that normal RAM is non-volatile [5, 8]. Although this is unacceptable for a production system, such pretending is fine for performance experiments.

A central purpose of this paper is to promote timing-accurate storage emulation as a first-class tool in the storage research toolbox. Towards this end, we describe its unique capabilities, demonstrate its relatively straightforward realization, and illustrate its power with several experiments that we could not otherwise perform.

3 Emulator design

A timing-accurate storage emulator must appear to its host system to be the storage subsystem that it emulates. Doing so involves three main tasks. First, the emulator must correctly support the protocols of the interface behind which it is implemented. Second, the emulator must complete requests in the amount of time computed by a model of the storage subsystem. Third, the emulator must retain copies of written data to satisfy read requests. This section describes how these three tasks are handled and the steps an emulator goes through to service storage requests.

3.1 Emulator components

Figure 3 shows the internals of a timing-accurate storage emulator. This section describes how the components of the emulator work to satisfy three tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and overflow storage).

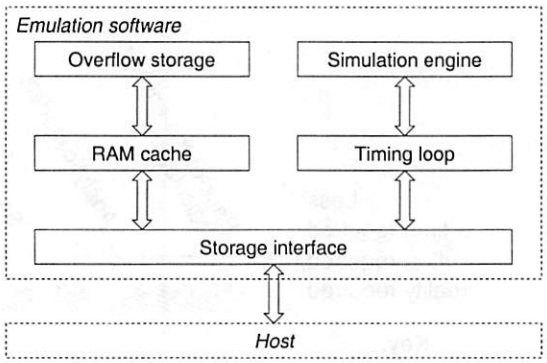


Figure 3: ***Emulation software internals.*** *The five components inside the "storage emulation software" box comprise the three primary emulator tasks: communications management (the storage interface), timing management (the simulation engine and timing loop), and data management (the RAM cache and overflow storage).*

Communications management. The storage interface component connects the emulator to the host system. As such, it must export the proper interface. The storage interface ensures that requests are transferred to and from the host according to the emulated protocol. Incoming requests are parsed and passed to the other emulator components, and outgoing messages are properly formatted for return to the host. In addition to servicing requests, the storage interface must respond appropriately to exceptional cases such as malformed requests or device errors.

In response to a read or write request, the storage interface parses the request, checks its validity, and then passes it to the timing and data management components of the emulator. In some cases, it may have to interact further with the host (e.g., for bus arbitration or if the emulated device supports disconnection). In addition to reads and writes, the emulator must support control requests that return information about the emulated drive such as its capacity, status, or error condition. In practice, a subset of often-used control commands usually suffices. When a request is completed, the response is formatted appropriately for the emulated protocol and forwarded to the host through the storage interface.

Timing management. The simulation engine and timing loop work together to provide the timing-accurate nature of the emulation. Specifically, the simulator determines how long each request should take to complete, and the timing loop ensures that completion is reported after the determined amount of time.

There are two ways that the simulation engine and timing loop can interact. One approach keeps the two separate: when a request arrives, the timing loop calls the simulator code once to get the service time. In this approach, the simulator code takes the real-world arrival time and the request details, and it returns the computed service time. After the appropriate real-time delay, the timing loop tells the storage interface component to report completion. The emulator-based evaulation of eager writing [25] used a disk simulator by Kotz et al. [15] in this way.

Although it is straightforward, this first approach often does not properly handle concurrent requests. For example, a new request arrival may affect the service time of outstanding requests due to bus contention, request overlapping, or request scheduling. A more general approach is to synchronize the advancement of the simulator's internal clock with the real-world clock. This synchronization can most easily be done with event-based simulation.

An event-based simulator breaks each request into a series of abstract and physical events: REQUEST ARRIVAL, CONTROLLER THINK TIME COMPLETE, DISK SEEK COMPLETE, READ OF SECTOR *N* COMPLETE, and so on. Each event is associated with a time, and an event "occurs" when the simulator's clock reaches the corresponding time. Event occurrences are processed by simulation code that updates state and schedules subsequent events. For example, the CONTROLLER THINK TIME COMPLETE event may be scheduled to occur a constant time after the REQUEST ARRIVAL event.

To synchronize an event-based simulation with the real world, the emulator lets the timing loop control the simulator clock advancement. When each event completes, the simulator engine notifies the timing loop of the next

scheduled event time. The timing loop waits until that time arrives, then calls back into the simulator to begin processing the next event. If a new request arrives, a REQUEST ARRIVAL event is prepended to the simulator's event list with the current wall clock time, and the timing loop calls back into the simulator immediately. When the REQUEST COMPLETE event ultimately occurs, the simulator engine notifies the storage interface.

In practice, the request arrival and completion times must be skewed slightly to account for processing and communication delays. The arrival time of a request is adjusted backwards slightly to account for the delay in receiving the request. Likewise, the simulator runs slightly ahead of the real-world clock so that the storage interface will start sending completion messages early enough for them to arrive on time. An obvious additional requirement is that the simulation computations themselves be fast enough that they do not delay completion messages; the computation time for any given request must be lower than the computed service time.

Data management. In addition to providing accurate timing of requests, emulation software must provide a consistent view of stored data. This is satisfied by the combination of a RAM-based block (sector) cache and overflow storage for swapping blocks from the cache. These components act as a conventional memory manager: groups of blocks can be grouped into "pages" that are evicted from or promoted into the cache. The overflow storage is only necessary for workloads requiring active storage in excess of the memory allocated to the emulation software. Possible implementations of the overflow storage include paging to one or more locally-attached disk drives, or paging to shared network-based RAM [2].

Data transfers from overflow storage may not complete quickly enough when emulating a high-performance device. When this is the case, cache preloading schemes may be necessary to ensure high RAM cache hit rates. These schemes can take advantage of the repeatability of experiments. For example, a workload could be initially run solely to generate a trace of accessed blocks, then run a second time using that trace to intelligently preload the cache throughout execution.

Since a timing-accurate storage emulator is used only as a performance evaluation tool and not as a production data store, persistence characteristics can be relaxed to increase performance. For example, write-back caching can be used to avoid costly overflow storage delays. If the system crashes and data is lost, the experiment can simply be re-run.

3.2 Host system interactions

Figure 4 shows the two most natural points at which to integrate a storage emulator into a host system. In the first (*local* emulation), the device driver is modified to communicate directly with the emulation software rather than with real storage components. Although this does involve some modifications to the system under test, they are restricted to the device driver. In the second (*remote* emulation), the host system is left unmodified and the emulation software runs on a second computer attached to the host via a storage interconnect. The second computer responds just as a real storage device would. Both integration points leave intact the application and OS software which is doing the real work and generating storage requests. Both also share a 3-step interface between the emulation software and the rest of the system.

Step 1: Send the request to the emulator. When a read or write request arrives at the device driver, it is directed to the emulated device. In the case of local emulation, the device driver is modified to be aware of the emulation software and explicitly delivers the request to it. A device that is emulated remotely does not need a modified device driver; requests are sent unmodified across the bus to the emulation machine which in turn delivers the request to the emulation software located there. Once the emulation software (either local or remote) has the request, it issues it to the simulator engine to determine how long the request should take to complete.

Step 2: Transfer data between the host and emulator. The emulation software initiates the data transfer. In the case of a read request, data is transferred from the RAM cache to the host. In the case of a write request, data goes from the host into the RAM cache and is saved to service future reads. Data transfer should usually begin soon after the request arrives, since all data must be transferred before the completion time computed by the simulator in Step 1. A local emulator can pass pointers to buffers in its RAM cache directly to the modified device driver. The driver then copies data between these userspace buffers and the appropriate kernel buffers. A remote emulator sends data over the bus to the host.

Step 3: Send the reply to the device driver. The emulation software waits until the request service time as determined in Step 1 elapses. At this point, a completion interrupt must be delivered to the OS. In the remote case, the completion message is sent over the bus, just as with a normal storage device, and the unmodified device driver deals with it appropriately. In the local case, the emulation software directly notifies the device driver that the request is complete at the device level. The driver then calls back into the operating system to complete the request at the system level.

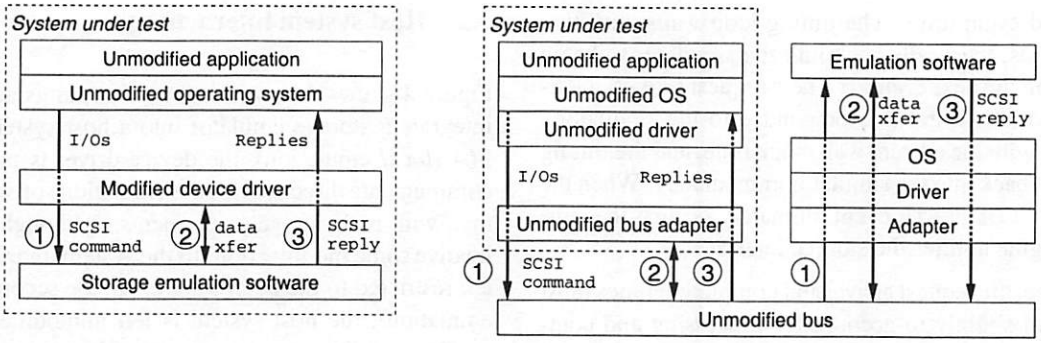


Figure 4: ***Communication paths when emulation is run (a) locally or (b) remotely.*** *When run locally, emulation software communicates directly with a modified device driver in the kernel. Under remote emulation, all modifications take place outside the system under test, eliminating any impact of the emulation overheads. The three steps are described in Section 3.2.*

The local design works well in practice and allows for extra communication paths between the operating system and emulator. For example, the device driver can measure perceived request service times and communicate these to the emulator, enabling the emulator to refine its model of communications overheads. In addition, this architecture enables evaluation of nonstandard device interfaces (such as freeblock requests or exposed eager writes) as discussed in Section 6.3.

However, a local emulator will have some impact on the system under test. Device driver modifications are necessary for communications with the emulator, and extra CPU time and memory are used to run the emulation software, which could perturb the host's workload. Using a dual-processor machine with one CPU dedicated to emulation and with added memory dedicated to the RAM cache can mitigate this overhead, but some interference is inevitable. A remote emulator avoids these perturbations completely by performing the emulation on separate, dedicated hardware. In this case, host overheads are eliminated and no modifications are required in the host's device driver.

In addition to device-specific delays, a local emulator must account for bus delays, since there is no physical bus between the host and the emulator. An advantage of this is that it allows emulation of devices "connected" to very fast local buses (for example, the PCI or system bus) or even emulation of the interconnect itself. A remote emulator that is physically attached to the host via a bus need not calculate such delays, unless it is emulating a different storage interconnect.

4 Implementation of the Memulator

This section describes the implementation of the *Memulator*, a prototype timing-accurate storage emulator. The emulation software runs as a user-level application and communicates with the host via a modified SCSI device driver. The Memulator can be run as either a local or remote emulator.

User-level emulation software. In the Memulator, user-level emulation software does the core work of timing-accurate storage emulation. It interprets requests, retains stored data, simulates device timings, and sends replies after the correct delays. This component is common to both the local and remote Memulator.

Timings are computed by DiskSim, which is an event-driven storage simulator [10]. The Memulator interacts with DiskSim via its "external control mode," in which external software (the timing loop) calls into DiskSim, specifies how far the simulation time should proceed before control is returned, and is then told when the next DiskSim-internal event should happen. The timing loop keeps the simulation time in close proximity to the real-world clock as given by the processor's time stamp counter.

Main memory is used to hold data written by previous requests. The Memulator's RAM cache is allocated and pinned in-core in its entirety during initialization using the `malloc()` and `mlock()` system calls. The operating system's resource limits may need to be adjusted to allow the pinning of a large memory region. The Memulator does not currently support overflow storage, so the working set of each experiment is limited to the cache's capacity.

When invalid opcodes, out-of-range requests, or invalid target/LUN pairs are received, the Memulator's storage

interface generates the appropriate sense code and immediately returns an error condition. Table 1 lists the SCSI commands supported by the Memulator. These commands are sufficient to allow Linux or FreeBSD to mount and use Memulator devices as SCSI disks.

Local emulation. The local version of the Memulator runs on the Linux 2.4 operating system. When used for local emulation, the user-level emulation software runs on the system under test as illustrated in Figure 4(a).

The modified device driver is a low-level component in the Linux SCSI subsystem. The driver accepts SCSI requests (`Scsi_Cmnd` structures) from the Linux kernel via the standard mid-to-low-level `queuecommand()` interface and passes them on to the storage interface as described below. When a request is complete, the driver notifies the kernel using the standard `scsi_done()` mid-level callback.

The Memulator's storage interface communicates with the device driver via modified system calls on the special character device file `/dev/memulator`. The storage interface uses the `poll()` system call to discover that a new request has arrived at the driver. It then uses the `read()` system call to transfer the 6-, 10-, or 12-byte SCSI command, the target device number, the logical unit number, and a unique request identifier. Following this transfer the timing loop immediately prepends a REQUEST ARRIVAL event to the DiskSim event queue. The arrival timestamp is skewed approximately 25 μs into the past to account for communications overheads; this value was determined empirically for our experimental setup. Once the newly arrived request is enqueued, the storage interface immediately directs the device driver to copy the requested data between the user and kernel memory buffers. The emulation software later uses the `write()` system call to notify the device driver when it determines the request is complete.

Table 1: ***Required SCSI command support.*** *Support for the upper six commands is necessary for an emulator to interact with a Linux 2.4 host. All nine commands must be supported when communicating with a FreeBSD 4.4 host.*

Command	Function
READ (6 and 10)	Read data from device
WRITE (6 and 10)	Write data to device
TEST UNIT READY	Check for device online
INQUIRY	Get device parameters
READ CAPACITY	Get device size in sectors
REQUEST SENSE	Get details of last error
MODE SENSE	Configure device
WRITE AND VERIFY	Verify data on device
SYNCHRONIZE CACHE	Flush device cache

Remote emulation. The remote version of the Memulator runs on the FreeBSD 4.4 operating system. When used for remote emulation, the Memulator runs entirely on a separate computer system. Both the host and remote systems are connected to a shared storage interconnect as illustrated in Figure 4(b).

Remote emulation requires hardware support in the bus adapter to act as a target in order to receive commands from an initiator. The operating system must also handle incoming requests and direct them to the user-level emulation software. This support is provided by FreeBSD's CAM subsystem when used with certain SCSI or Fibre Channel cards. The storage interface communicates with a modified target mode device driver in much the same manner as described for the local version of the Memulator. For remote emulation experiments the arrival timestamp is skewed approximately 120 μs into the past.

Alternative implementations of a remote emulator could leverage storage networking protocols such as iSCSI or run on dedicated custom hardware connected to the PCI or system buses of the host system.

5 Memulator validation

This section presents three evaluations of the Memulator. First, we show that the upper bound of performance for our setup is sufficient to meet the requirements of the devices we model. Second, we show that the Memulator accurately reflects the timings of a simulated storage device. Third, we show that this timing-accuracy can translate into accurate emulation of a real storage device.

5.1 Experimental setup

Local emulation experiments are performed on a 700 MHz dual-processor Intel Pentium III-based workstation with 2 GB RAM running Linux 2.4.2. The second CPU is used to reduce interference between the emulation software and the regular workload. During all experiments, 1,792 MB of main memory is pinned as the Memulator's RAM cache, leaving 256 MB for the "real" system activity. Unless otherwise specified, the experiments in this paper are run under local emulation.

For remote emulation, the Memulator runs on a single-processor workstation with 2 GB RAM and FreeBSD 4.4. The host system is a single-processor workstation with 256 MB RAM running either Linux or FreeBSD as noted below. The host and remote systems are connected by an 80 MB/s SCSI bus via Adaptec AHA-29160 adapters.

The disk used for comparison is the Seagate Cheetah X15, a 15,000 RPM disk with 3.9 ms average seek time and 18 GB capacity. It is connected to a 1 Gbit/s Fibre Channel network (FC-AL) hosted by a QLogic ISP2100.

This disk was chosen as a reasonable example of a modern high-end disk. Validated DiskSim specifications are available [7] for this disk, allowing us to configure the Memulator using validated parameters for the Cheetah X15.

To focus on storage performance, we use six artificial workloads: "random or mixed" crossed with "small, uniform, or large." A *random* workload has zero probability of local access or sequential access; request starting locations are uniformly distributed across the storage capacity. A *mixed* workload has 30% probability of "local" access (within 500 LBNs of the previous request) and 20% probability of sequential access. A *small* workload is composed of 8-sector (4KB) requests, a *large* workload uses 256-sector (128KB) requests, and a *uniform* workload has uniformly distributed request sizes in intervals of 2KB over the range [2KB, 130KB]. Therefore, a "mixed large" workload has some sequential and local accesses and is composed of 128KB requests. All workloads are made up of 1,000 I/O requests, of which 67% are reads.

We also present results for the PostMark benchmark [14]. PostMark was designed to measure the performance of a file system used for electronic mail, news, and web-based services. It creates a large number of small files, on which a specified number of transactions are performed. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The transaction types are chosen randomly with consideration given to user definable weights. The benchmark parameters in these experiments specify 20,000 transactions on 10,000 files, with a file size of between 10KB and 20KB.

5.2 Implementation performance

To determine the fastest device that the Memulator can emulate, we configured it to reply with request completions immediately after the data transfer phase. By removing the timing component, all that remains is the overhead required for emulation.

We measure request rate and bandwidth for both local and remote configurations. Request rate is measured by issuing 2^{10} one-sector read requests and dividing the number of requests by the elapsed time. Bandwidth is measured by issuing 2^{10} 1024-sector read requests and dividing the total bytes transferred by elapsed time. System-level caching is disallowed; all requests are synchronously issued through the Linux SCSI generic (SG) interface in local emulation and the FreeBSD direct access interface during remote emulation.

The results for local and remote emulation are shown in Table 2, along with the required performance values for MEMS-based storage [11] and the Seagate Cheetah X15 [23]. Both the local and remote Memulator configurations achieve the required performance threshold for both devices along both axes.

5.3 Memulator accuracy

To evaluate how closely the Memulator comes to perfect timing-accurate emulation, we execute the six artificial workloads against both the Memulator and against standalone DiskSim. We run these workloads against the Memulator by generating a series of SCSI requests based on each workload's characteristics and issuing the requests to the Memulator through the Linux SCSI generic (SG) interface. The SG interface allows an application to create SCSI requests at the user level, pass these commands directly to the device driver, receive replies from the driver, and handle them directly at user level. We measure the Memulator's accuracy by taking arrival and completion timestamps inside the device driver, then comparing these times to the per-request times reported by standalone DiskSim. If these times match, the kernel is seeing exactly the performance intended.

Table 3 displays the results, and Figure 5 provides a supplementary view of the uniform workloads. The average |% emulation error| (that is, the average per-request error, independent of whether the request completed too fast or too slow) is less than 0.33% in all cases, and over 99% of all requests have less than 2% error. Most errors larger than 2% are only slightly larger. Exceptions fall into two categories: (1) the simulator can take too long to compute a result, and (2) the emulation program can be context-switched off of the CPU. These exceptions occurs for about one request in 1000 in the experiments. Fundamentally, the extra delays from both categories are unbounded, but we have observed only 5–10% inaccuracy from the first and up to 3–4ms errors from the second.

Table 2: ***Upper bound of Memulator performance.*** *The local and remote emulation values are measured with the Memulator configured to return data and replies as quickly as possible. The remote emulation experiments are run using a FreeBSD host. The MEMS-based storage [11] and Seagate Cheetah X15 [23] values represent peak bandwidth and average request rate for random I/O.*

	Request rate	Bandwidth
Available performance		
Local emulation	22,468req/s	84.5MB/s
Remote emulation	8,883req/s	103.5MB/s
Required performance		
MEMS-based storage	1,422req/s	76MB/s
Cheetah X15 disk	244req/s	49MB/s

Table 3: ***Memulator accuracy.*** *Each workload represents 1,000 requests as measured inside the device driver and inside DiskSim. "Mean service time" is the average request service time reported by the simulation engine. "Mean emulation error" reports the average difference between the measured (emulated) time and the simulated service time of each request. Negative values represent requests that finished more quickly than the simulated time. "Mean |emulation % error|" is the average of the absolute values of percent error of the emulated time for each request with respect to the simulated service time. "Requests under 1% error" shows the percentage of requests completing within 1% of their simulated time.*

	small requests (4 KB)		**uniform** (2–130 KB)		**large requests** (128 KB)	
	random	mixed	random	mixed	random	mixed
10ms constant time model						
mean service time	10,000 μs	10,000 μs	10,000 μs	10,000 μs	10,000 μs	10,000 μs
mean emulation error	-0.65 μs	-0.49 μs	0.93 μs	0.75 μs	2.36 μs	1.42 μs
mean |emulation % error|	0.01%	0.01%	0.04%	0.02%	0.05%	0.05%
requests under 1% error	100%	100%	99.9%	100.0%	100.0%	99.9%
Cheetah X15 model						
mean service time	6,509 μs	5,568 μs	9,116 μs	8,410 μs	11,301 μs	10,154 μs
mean emulation error	5.33 μs	10.87 μs	0.92 μs	-0.19 μs	-10.08 μs	-9.70 μs
mean |emulation % error|	0.11%	0.33%	0.07%	0.08%	0.09%	0.11%
requests under 1% error	99.5%	92.4%	99.9%	99.1%	100%	100%
MEMS-based storage model						
mean service time	1,057 μs	1,049 μs	2,001 μs	1,957 μs	2,846 μs	2,857 μs
mean emulation error	-0.21 μs	0.16 μs	-0.59 μs	-0.50 μs	-1.54 μs	-0.01 μs
mean |emulation % error|	0.13%	0.16%	0.13%	0.14%	0.13%	0.11%
requests under 1% error	100%	99.4%	100%	98.9%	100%	100%

Table 4: ***Application run times using the Memulator vs. using a real disk.*** *Both the local and the remote Memulator faithfully reproduce the performance of the disk under the PostMark benchmark, but there is some interference with application performance in the local emulation case. PostMark was configured to use 10,000 files between 10 KB and 20 KB in size, with 20,000 transactions. Each data point is the average of ten runs of the benchmark. The remote emulation experiments are run using a Linux host.*

	Real Cheetah X15	Local emulation	Remote emulation
PostMark			
run time	78.422 s	74.523 s	78.532 s
standard deviation	0.375	0.618	0.244
percent error from real disk	—	4.97%	0.14%

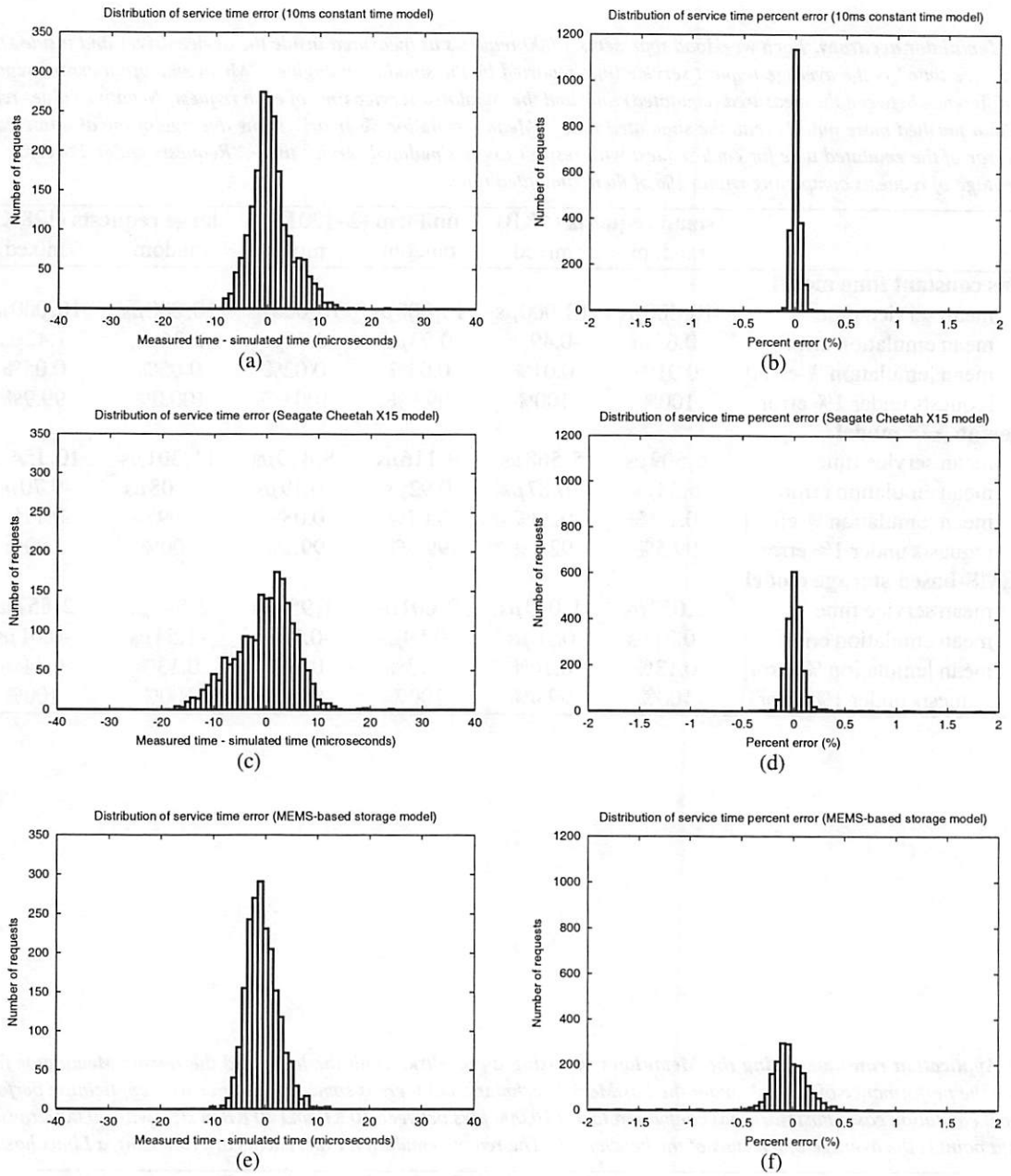


Figure 5: ***Densities of emulation error and percent error.*** *Emulation error is defined as the difference between the time reported by the emulator and that reported by the simulator alone running the same workload. This error shows the timing discrepancy caused by variation in the overheads of emulation, such as passing commands and data to the emulation software. A perfect emulator would introduce no discrepencies and the times would match exactly. The Memulator is shown to introduce only minor discrepencies compared to DiskSim running alone. Each graph shows the combined results of the "random uniform" and "mixed uniform" workloads, for a total of 4,000 requests. Percent error is calculated with respect to the simulated request time. Bin sizes are 1 µs in the service time graphs and 0.05% in the percent error graphs.*

Table 5: ***Exploring a change to disk firmware.*** *Here we use timing-accurate storage emulation to add zero-latency access capability to a disk (the Seagate Cheetah X15) that in reality does not support it. Each data point is the average of ten runs of the benchmark.*

	Default	Zero-latency	Decrease in time
PostMark			
run time	74.523 s	74.469 s	0.1%
std. dev.	0.618	0.783	—

Table 6: ***Exploring technology trends.*** *These results show the effect of scaling the rotation speed of the Seagate Cheetah X15 to 30,000 RPM. Each data point is the average of ten runs of the benchmark.*

	15,000 RPM	30,000 RPM	Decrease in time
PostMark			
run time	74.523 s	66.215 s	11.1%
std. dev.	0.618	0.651	—

5.4 Comparison with real disks

Having established in Section 5.3 that the Memulator matches its internal simulation timings, we compare application run times using the Memulator vs. using real disks. The results are shown in Table 4. In this experiment, we see the advantage of using remote emulation rather than local emulation. Interactions between the "real" application/OS activity and local emulation activity cause the benchmark runtime to be off by 5%. With remote emulation, on the other hand, benchmark performance is almost identical (within 0.14%) whether using the disk or the Memulator. Although the close match for remote emulation is comforting, it is important to remember that the Memulator's main responsibility is to ensure fidelity to the model's timing. It is the responsibility of the model's creator to ensure fidelity to the modeled device.

6 Memulator-enabled experiments

This section illustrates the power of timing-accurate storage emulation by describing experiments made possible by the Memulator. These experiments fall into three categories: experiments with modified disks, experiments with futuristic devices, and experiments with storage interface extensions.

6.1 Changes to existing devices

A long-standing obstacle for most experimental storage researchers is that disk firmware source code is unavailable. This prevents direct experimentation with modifications to firmware algorithms, including LBN-to-physical mapping, on-board cache management, prefetching, and scheduling. With the Memulator, this obstacle is partially removed.

To illustrate the new capability, we compare application performance when a disk has zero-latency read support and when it does not. Zero-latency read (a.k.a. read-on-arrival and immediate read) allows the disk firmware to fetch sectors from the media in any order, rather than requiring strictly ascending LBN order. When exactly one track is fetched, zero-latency read support allows the media transfer to begin as soon as the seek is complete; since every sector on the track is desired, the media transfer requires at most one rotation. Without zero-latency read, the same request would suffer the normal rotational latency before the one rotation of media transfer.

Table 5 shows the performance impact of zero-latency reads on the PostMark benchmark described in the previous section. Although some disks support zero-latency reads, the Cheetah X15 does not. This design choice is correct for PostMark: as the workload involves mostly small files and background disk writes, there is little opportunity to benefit from zero-latency reads. A workload with larger transfers could be expected to benefit.

Although this particular result may not be interesting, the ability to conduct the experiment is. Enabling full system experimentation may increase the believability of results pertaining to future firmware enhancement proposals.

In addition to firmware and algorithmic changes, timing-accurate storage emulation enables experiments reflecting hardware and technology changes. For example, Table 6 shows the performance impact of doubling the Cheetah X15's rotational speed. Despite reducing rotational latency by half and doubling the media transfer rate, this hardware upgrade results in only an 11.1% improvement for PostMark.

6.2 New storage technologies

Microelectromechanical systems (MEMS)-based storage is an exciting new technology that could soon be available in systems. MEMS are very small scale mechanical structures—on the order of 10–1000 μm—fabricated on the surface of silicon chips [26]. Using thousands of minute MEMS read/write heads, data bits can be stored in and retrieved from media coated on a small movable media sled [4, 11, 24]. With higher storage densities (260–720 Gbit/in^2) and lower random access times (under 1 ms), MEMS-based storage devices could play a significant role in future systems.

The Memulator allows us to explore the impact of using MEMS-based storage in existing computer systems, even

Table 7: ***MEMS-based storage vs. Seagate Cheetah X15.*** *These results compare the runtime of the PostMark benchmark on a Cheetah X15 disk and a MEMS-based storage device. Each data point is the average of ten runs of the benchmark.*

	Cheetah X15	MEMS-based	Decrease in time
PostMark			
run time	74.523 s	51.420 s	31.0%
std. dev.	0.618	1.678	—

though the devices themselves are several years from production. We configured the Memulator to use the G2 device described by Schlosser et al. [22].

Table 7 compares the performance of PostMark running on the Cheetah X15 and on MEMS-based storage. Although the average response time of the disk was five times greater than the MEMS-based storage device (7.91 vs. 1.59 ms), we observed only a 31% decrease in overall runtime when using MEMS-based storage. This is because of the relatively small dataset (only 10,000 10–20 KB files) and the aggressive writeback caching performed by the Linux host system. This caching masks much of the benefit of the faster I/O times of MEMS-based storage. For workloads with larger data sets or synchronous writes (e.g., transaction processing), the overall system performance improvements would be greater. The performance with MEMS-based storage approaches our setup's minimum runtime of 48.512 s, which we measured by rerunning the experiment with the Memulator configured to respond to all I/O requests immediately.

6.3 Storage interface extensions

A third set of storage designs that would benefit from emulation-based evaluation involves storage interface extensions. Such extensions require that both the host OS and the storage device be modified to utilize a new interface. Not only must the interface be supported, but often the implementations of both sides must change to truly exploit a new interface's potential. Two examples of this arise from recently-proposed mechanisms: freeblock scheduling [16] and eager writing [25].

Freeblock scheduling consists of replacing the rotational latency delays of high-priority disk requests with background media transfers. Since the high-priority data will rotate around to the disk head at the same time, regardless of what is done during the rotational latency, these background media transfers can occur without slowing the high-priority requests. It is believed that freeblock scheduling can be accomplished most effectively from within disk firmware. Before they will consider new functionality, however, disk manufacturers want to know exactly what the interface should be and what real application environments will benefit. Since researchers have no access to disk firmware, this creates a chicken-and-egg problem. The Memulator, combined with OS source code (e.g., Linux), enables the interface and application questions to be explored.

Eager writing consists of writing new data to an unused location near the disk head's current location. Such dynamic data placement can significantly reduce service times. As with freeblock scheduling, the best decisions would probably be made from within disk firmware. However, this approach would require the firmware to maintain a mapping table, and it would not benefit from the OS's knowledge of high-level intra-file and inter-file data relationships. A more cooperative interface might allow the host system to direct the disk to write a block to any of several locations (whichever is most efficient); the device would then return the resulting location, which could be recorded in the host's metadata structures. Difficulties would undoubtedly arise with this design, and the Memulator enables OS prototyping and experimentation to flesh them out.

7 Summary

This paper describes and promotes timing-accurate storage emulation as a foundation for more thorough evaluation of proposed storage designs. Measurements of our prototype, the Memulator, demonstrate that 99% of its response times are within 2% of their simulator-computed targets. More importantly, the Memulator allows us to run real application benchmarks on real systems equipped with storage components that we cannot yet build, such as disks with firmware extensions and MEMS-based storage.

Acknowledgments

We thank our shepherd Darrell Long, the anonymous reviewers, Andy Klosterman, and Liddy Shriver for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including EMC, HP, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. We thank Justin Gibbs for assistance in configuring target mode under FreeBSD. John Griffin is supported in part by a National Science Foundation Graduate Fellowship. Steve Schlosser is supported by an Intel Graduate Fellowship.

References

[1] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: emulation and experiment. *ACM SIGCOMM Conference* (Cambridge, MA, 28 August–1 September, 1995). Published as *Computer Communication Review*, **25**(4):185–195. ACM, 1995.

[2] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

[3] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A virtual machine emulator for performance evaluation. *Communications of the ACM*, **23**(2):71–80. ACM, February 1980.

[4] L. Richard Carley, James A. Bain, Gary K. Fedder, David W. Greve, David F. Guillou, Michael S. C. Lu, Tamal Mukherjee, Suresh Santhanam, Leon Abelmann, and Seungook Min. Single-chip computers with microelectromechanical systems-based magnetic memory. *Journal of Applied Physics*, **87**(9):6680–6685, 1 May 2000.

[5] Peter Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: surviving operating system crashes. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):74–83, 1996.

[6] Kevin Fall. Network emulation in the Vint/NS simulator. *IEEE Symposium on Computers and Communications* (Red Sea, Egypt, 6–8 July 1999), pages 244–250, 1999.

[7] Greg Ganger and Jiri Schindler. Database of validated disk parameters for DiskSim. http://www.ece.cmu.edu/~ganger/disksim/diskspecs.html.

[8] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 49–60. Usenix Association, 14–17 November 1994.

[9] Gregory R. Ganger and Yale N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, **47**(6):667–678, June 1998.

[10] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim simulation environment version 1.0 reference manual*, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[11] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Modeling and performance of MEMS-based storage devices. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 17–21 June 2000). Published as *Performance Evaluation Review*, **28**(1):56–65, 2000.

[12] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Operating system management of MEMS-based storage devices. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 227–242. USENIX, 2000.

[13] David Hitz. *An NFS file server appliance*. Technical report. Network Appliance, August 1993.

[14] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[15] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. *A detailed simulation model of the HP 97560 disk drive*. Technical report PCS–TR94–220. Department of Computer Science, Dartmouth College, July 1994.

[16] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[17] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: a virtual workstation. *Annual USENIX Technical Conference* (New Orleans, LA, 15–19 June 1998), pages 119–30. USENIX Association, 1998.

[18] Brian D. Noble, M. Satyanarayanan, Giao T. Nguyen, and Randy H. Katz. Trace-based mobile network emulation. *ACM SIGCOMM Conference* (Cannes, France, 14–18 September 1997), pages 51–61, 1997.

[19] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM. Transactions on Modeling and Computer Simulation*, **7**(1):78–103. ACM, January 1997.

[20] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[21] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993.

[22] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 12–15 November 2000). Published as *Operating Systems Review*, **34**(5):1–12, 2000.

[23] Seagate Technology, Inc. *Cheetah X15 FC Disc Drive ST318451FC/FCV Product Manual, Volume 1*, Publication number 83329486, Rev. A, June 1, 2000.

[24] P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The "Millipede"—more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, **44**(3):323–340, 2000.

[25] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, 1999.

[26] Kensall D. Wise. Special issue on integrated sensors, microactuators, and microsystems (MEMS). *Proceedings of the IEEE*, **86**(8):1531–1787, August 1998.

Venti: a new approach to archival storage

Sean Quinlan and Sean Dorward
Bell Labs, Lucent Technologies

Abstract

This paper describes a network storage system, called Venti, intended for archival data. In this system, a unique hash of a block's contents acts as the block identifier for read and write operations. This approach enforces a write-once policy, preventing accidental or malicious destruction of data. In addition, duplicate copies of a block can be coalesced, reducing the consumption of storage and simplifying the implementation of clients. Venti is a building block for constructing a variety of storage applications such as logical backup, physical backup, and snapshot file systems.

We have built a prototype of the system and present some preliminary performance results. The system uses magnetic disks as the storage technology, resulting in an access time for archival data that is comparable to non-archival data. The feasibility of the write-once model for storage is demonstrated using data from over a decade's use of two Plan 9 file systems.

1. Introduction

Archival storage is a second class citizen. Many computer environments provide access to a few recent versions of the information stored in file systems and databases, though this access can be tedious and may require the assistance of a system administrator. Less common is the ability for a user to examine data from last month or last year or last decade. Such a feature may not be needed frequently, but when it is needed it is often crucial.

The growth in capacity of storage technologies exceeds the ability of many users to generate data, making it practical to archive data in perpetuity. Plan 9, the computing environment that the authors use, includes a file system that stores archival data to an optical jukebox [16, 17]. Ken Thompson observed that, for our usage patterns, the capacity of the jukebox could be considered infinite. In the time it took for us to fill the jukebox, the improvement in technology would allow us to upgrade to a new jukebox with twice the capacity.

Abundant storage suggests that an archival system impose a *write-once* policy. Such a policy prohibits either a user or administrator from deleting or modifying data once it is stored. This approach greatly reduces the opportunities for accidental or malicious data loss and simplifies the system's implementation.

Moreover, our experience with Plan 9 is that a write-once policy changes the way one views storage. Obviously, some data is temporary, derivative, or so large that it is either undesirable or impractical to retain forever and should not be archived. However, once it is decided that the data is worth keeping, the resources needed to store the data have been consumed and cannot be reclaimed. This eliminates the task of periodically "cleaning up" and deciding whether the data is still worth keeping. More thought is required before storing the data to a write-once archive, but as the cost of storage continues to fall, this becomes an easy decision.

This paper describes the design and implementation of an archival server, called Venti. The goal of Venti is to provide a write-once archival repository that can be shared by multiple client machines and applications. In addition, by using magnetic disks as the primary storage technology, the performance of the system approaches that of non-archival storage.

2. Background

A prevalent form of archival storage is the regular backup of data to magnetic tape [15]. A typical scenario is to provide backup as a central service for a number of client machines. Client software interfaces with a database or file system and determines what data to back up. The data is copied from the client to the tape device, often over a network, and a record of what was copied is stored in a catalog database.

Restoring data from a tape backup system can be tedious and error prone. The backup system violates the access permission of the file system, requiring a system administrator or privileged software to perform the task. Since they are tedious, restore operations are infrequent and problems with the process may go undetected. Potential sources of error abound: tapes are mislabeled

or reused or lost, drives wander out of alignment and cannot read their old tapes, technology becomes obsolete.

For tape backup systems, a tradeoff exists between the performance of backup and restore operations [1]. A full backup simplifies the process of restoring data since all the data is copied to a continuous region on the tape media. For large file systems and databases, incremental backups are more efficient to generate, but such backups are not self-contained; the data for a restore operation is scattered across multiple incremental backups and perhaps multiple tapes. The conventional solution is to limit the extent of this scattering by performing a full backup followed by a small number of incremental backups.

File systems such as Plan 9 [16, 17], WAFL [5], and AFS [7] provide a more unified approach to the backup problem by implementing a snapshot feature. A snapshot is a consistent read-only view of the file system at some point in the past. The snapshot retains the file system permissions and can be accessed with standard tools (ls, cat, cp, grep, diff) without special privileges or assistance from an administrator. In our experience, snapshots are a relied-upon and frequently-used resource because they are always available and easy to access.

Snapshots avoid the tradeoff between full and incremental backups. Each snapshot is a complete file system tree, much like a full backup. The implementation, however, resembles an incremental backup because the snapshots and the active file system share any blocks that remain unmodified; a snapshot only requires additional storage for the blocks that have changed. To achieve reasonable performance, the device that stores the snapshots must efficiently support random access, limiting the suitability of tape storage for this approach.

In the WAFL and AFS systems, snapshots are ephemeral; only a small number of recent versions of the file system are retained. This policy is reasonable since the most recent versions of files are the most useful. For these systems, archival storage requires an additional mechanism such as tape backup.

The philosophy of the Plan 9 file system is that random access storage is sufficiently cheap that it is feasible to retain snapshots permanently. The storage required to retain all daily snapshots of a file system is surprisingly modest; later in the paper we present statistics for two file servers that have been in use over the last 10 years.

Like Plan 9, the Elephant file system [18] retains many versions of data. This system allows a variety of storage reclamation policies that determine when a version of a file should be deleted. In particular, "landmark" versions of files are retained permanently and provide an archival record.

3. The Venti Archival Server

Venti is a block-level network storage system intended for archival data. The interface to the system is a simple protocol that enables client applications to read and write variable sized blocks of data. Venti itself does not provide the services of a file or backup system, but rather the backend archival storage for these types of applications.

Venti identifies data blocks by a hash of their contents. By using a collision-resistant hash function with a sufficiently large output, it is possible to consider the hash of a data block as unique. Such a unique hash is called the *fingerprint* of a block and can be used as the address for read and write operations. This approach results in a storage system with a number of interesting properties.

As blocks are addressed by the fingerprint of their contents, a block cannot be modified without changing its address; the behavior is intrinsically write-once. This property distinguishes Venti from most other storage systems, in which the address of a block and its contents are independent.

Moreover, writes are idempotent. Multiple writes of the same data can be coalesced and do not require additional storage space. This property can greatly increase the effective storage capacity of the server since it does not rely on the behavior of client applications. For example, an incremental backup application may not be able to determine exactly which blocks have changed, resulting in unnecessary duplication of data. On Venti, such duplicate blocks will be discarded and only one copy of the data will be retained. In fact, replacing the incremental backup with a full backup will consume the same amount of storage. Even duplicate data from different applications and machines can be eliminated if the clients write the data using the same block size and alignment.

The hash function can be viewed as generating a universal name space for data blocks. Without cooperating or coordinating, multiple clients can share this name space and share a Venti server. Moreover, the block level interface places few restrictions on the

structures and format that clients use to store their data. In contrast, traditional backup and archival systems require more centralized control. For example, backup systems include some form of job scheduler to serialize access to tape devices and may only support a small number of predetermined data formats so that the catalog system can extract pertinent meta-data.

Venti provides inherent integrity checking of data. When a block is retrieved, both the client and the server can compute the fingerprint of the data and compare it to the requested fingerprint. This operation allows the client to avoid errors from undetected data corruption and enables the server to identify when error recovery is necessary.

Using the fingerprint of a block as its identity facilitates features such as replication, caching, and load balancing. Since the contents of a particular block are immutable, the problem of data coherency is greatly reduced; a cache or a mirror cannot contain a stale or out of date version of a block.

3.1. Choice of Hash Function

The design of Venti requires a hash function that generates a unique fingerprint for every data block that a client may want to store. Obviously, if the size of the fingerprint is smaller than the size of the data blocks, such a hash function cannot exist since there are fewer possible fingerprints than blocks. If the fingerprint is large enough and randomly distributed, this problem does not arise in practice. For a server of a given capacity, the likelihood that two different blocks will have the same hash value, also known as a collision, can be determined. If the probability of a collision is vanishingly small, we can be confident that each fingerprint is unique.

It is desirable that Venti employ a cryptographic hash function. For such a function, it is computationally infeasible to find two distinct inputs that hash to the same value [10]. This property is important because it prevents a malicious client from intentionally creating blocks that violate the assumption that each block has a unique fingerprint. As an additional benefit, using a cryptographic hash function strengthens a client's integrity check, preventing a malicious server from fulfilling a read request with fraudulent data. If the fingerprint of the returned block matches the requested fingerprint, the client can be confident the server returned the original data.

Venti uses the Sha1 hash function [13] developed by the US National Institute for Standards and Technology (NIST). Sha1 is a popular hash algorithm for many security systems and, to date, there are no known collisions. The output of Sha1 is a 160 bit (20 byte) hash value. Software implementations of Sha1 are relatively efficient; for example, a 700Mhz Pentium 3 can compute the Sha1 hash of 8 Kbyte data blocks in about 130 microseconds, a rate of 60 Mbytes per second.

Are the 160 bit hash values generated by Sha1 large enough to ensure the fingerprint of every block is unique? Assuming random hash values with a uniform distribution, a collection of n different data blocks and a hash function that generates b bits, the probability p that there will be one or more collisions is bounded by the number of pairs of blocks multiplied by the probability that a given pair will collide, i.e.

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b}.$$

Today, a large storage system may contain a petabyte (10^{15} bytes) of data. Consider an even larger system that contains an exabyte (10^{18} bytes) stored as 8 Kbyte blocks ($\sim 10^{14}$ blocks). Using the Sha1 hash function, the probability of a collision is less than 10^{-20}. Such a scenario seems sufficiently unlikely that we ignore it and use the Sha1 hash as a unique identifier for a block. Obviously, as storage technology advances, it may become feasible to store much more than an exabyte, at which point it maybe necessary to move to a larger hash function. NIST has already proposed variants of Sha1 that produce 256, 384, and 512 bit results [14]. For the immediate future, however, Sha1 is a suitable choice for generating the fingerprint of a block.

3.2. Choice of Storage Technology

When the Plan 9 file system was designed in 1989, optical jukeboxes offered high capacity with respectable random access performance and thus were an obvious candidate for archival storage. The last decade, however, has seen the capacity of magnetic disks increase at a far faster rate than optical technologies [20]. Today, a disk array costs less than the equivalent capacity optical jukebox and occupies less physical space. Disk technology is even approaching tape in cost per bit.

Magnetic disk storage is not as stable or permanent as optical media. Reliability can be improved with technology such as RAID, but unlike write-once optical disks, there is little protection from erasure due to failures of the storage server or RAID array firmware. This issue is discussed in Section 7.

Using magnetic disks for Venti has the benefit of reducing the disparity in performance between conventional and archival storage. Operations that previously required data to be restored to magnetic disk can be accomplished directly from the archive. Similarly, the archive can contain the primary copy of often-accessed read-only data. In effect, archival data need not be further down the storage hierarchy; it is differentiated by the write-once policy of the server.

4. Applications

Venti is a building block on which to construct a variety of storage applications. Venti provides a large repository for data that can be shared by many clients, much as tape libraries are currently the foundation of many centralized backup systems. Applications need to accommodate the unique properties of Venti, which are different from traditional block level storage devices, but these properties enable a number of interesting features.

Applications use the block level service provided by Venti to store more complex data structures. Data is divided into blocks and written to the server. To enable this data to be retrieved, the application must record the fingerprints of these blocks. One approach is to pack the fingerprints into additional blocks, called *pointer blocks*, that are also written to the server, a process that can be repeated recursively until a single fingerprint is obtained. This fingerprint represents the root of a tree of blocks and corresponds to a hierarchical hash of the original data.

A simple data structure for storing a linear sequence of data blocks is shown in Figure 1. The data blocks are located via a fixed depth tree of pointer blocks which itself is addressed by a root fingerprint. Applications can use such a structure to store a single file or to mimic the behavior of a physical device such as a tape or a disk drive. The write-once nature of Venti does not allow such a tree to be modified, but new versions of the tree can be generated efficiently by storing the new or modified data blocks and reusing the unchanged sections of the tree as depicted in Figure 2.

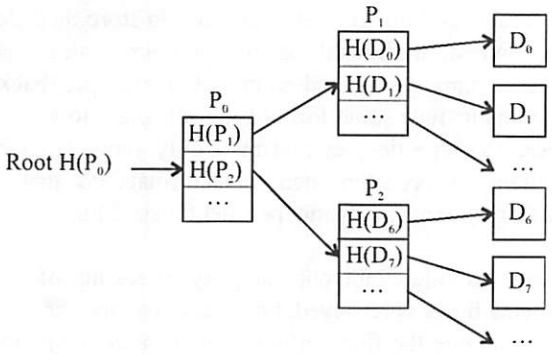


Figure 1. A tree structure for storing a linear sequence of blocks

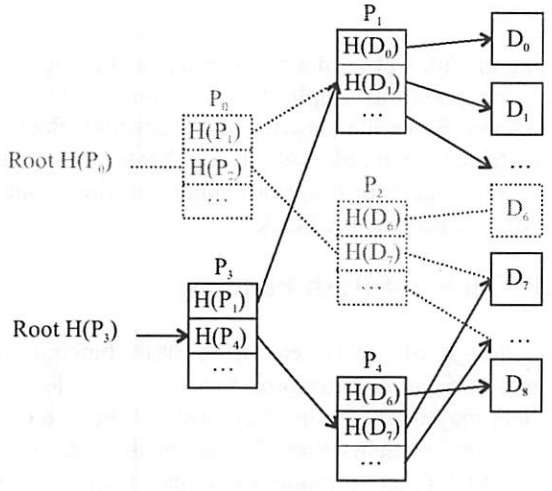


Figure 2. Build a new version of the tree.

By mixing data and fingerprints in a block, more complex data structures can be constructed. For example, a structure for storing a file system may include three types of blocks: directory, pointer, and data. A directory block combines the meta information for a file and the fingerprint to a tree of data blocks containing the file's contents. The depth of the tree can be determined from the size of the file, assuming the pointer and data blocks have a fixed size. Other structures are obviously possible. Venti's block-level interface leaves the choice of format to client applications and different data structures can coexist on a single server.

The following sections describes three applications that use Venti as an archival data repository: a user level archive utility called vac, a proposal for a physical level backup utility, and our preliminary work on a new version of the Plan 9 file system.

4.1. Vac

Vac is an application for storing a collection of files and directories as a single object, similar in functionality to the utilities tar and zip. With vac, the contents of the selected files are stored as a tree of blocks on a Venti server. The root fingerprint for this tree is written to a vac archive file specified by the user, which consists of an ASCII representation of the 20 byte root fingerprint plus a fixed header string, and is always 45 bytes long. A corresponding program, called unvac, enables the user to restore files from a vac archive. Naturally, unvac requires access to the Venti server that contains the actual data, but in most situations this is transparent. For a user, it appears that vac compresses any amount of data down to 45 bytes.

An important attribute of vac is that it writes each file as a separate collection of Venti blocks, thus ensuring that duplicate copies of a file will be coalesced on the server. If multiple users vac the same data, only one copy will be stored on the server. Similarly, a user may repeatedly vac a directory over time and even if the contents of the directory change, the additional storage consumed on the server will be related to the extent of the changes rather than the total size of the contents. Since Venti coalesces data at the block level, even files that change may share many blocks with previous versions and thus require little space on the server; log and database files are good examples of this scenario.

On many Unix systems, the dump utility is used to back up file systems. Dump has the ability to perform incremental backups of data; a user specifies a dump level, and only files that are new or have changed since the last dump at this level are written to the archive. To implement incremental backups, dump examines the modified time associated with each file, which is an efficient method of filtering out the unchanged files.

Vac also implements an incremental option based on the file modification times. The user specifies an existing vac file and this archive is used to reduce the number of blocks written to the Venti server. For each file, vac examines the modified time in both the file system and the vac archive. If they are the same, vac copies the fingerprint for the file from the old archive into the new archive. Copying just the 20-byte fingerprint enables the new archive to include the entire file without reading the data from the file system nor writing the data across the network to the Venti server. In addition, unlike an incremental dump, the resulting archive will be identical to an archive generated without the incremental option; it is only a performance improvement. This means there is no need to have multiple levels of backups, some incremental, some full, and so restore operations are greatly simplified.

A variant of the incremental option improves the backup of files without reference to modification times. As vac reads a file, it computes the fingerprint for each block. Concurrently, the pointer blocks of the old archive are examined to determine the fingerprint for the block at the same offset in the old version of the file. If the fingerprints are the same, the block does not need to be written to Venti. Instead, the fingerprint can simply be copied into the appropriate pointer block. This optimization reduces the number of writes to the Venti server, saving both network and disk bandwidth. Like the file level optimization above, the resulting vac file is no different from the one produced without this optimization. It does, however, require the data for the file to be read and is only effective if there are a significant number of unchanged blocks.

4.2. Physical backup

Utilities such as vac, tar, and dump archive data at the file or logical level: they walk the file hierarchy converting both data and meta-data into their own internal format. An alternative approach is block-level or physical backup, in which the disk blocks that make up the file system are directly copied without interpretation. Physical backup has a number of benefits including simplicity and potentially much higher throughput [8]. A physical backup utility for file systems that stores the resulting data on Venti appears attractive, though we have not yet implemented such an application.

The simplest form of physical backup is to copy the raw contents of one or mores disk drives to Venti. The backup also includes a tree of pointer blocks, which enables access to the data blocks. Like vac, the end result is a single fingerprint representing the root of the tree; that fingerprint needs to be recorded outside of Venti.

Coalescing duplicate blocks is the main advantage of making a physical backup to Venti rather than copying the data to another storage medium such as tape. Since file systems are inherently block based, we expect coalescing to be effective. Not only will backups of a file system over time share many unchanged blocks, but even file systems for different machines that are running the same operating system may have many blocks in common. As with vac, the user sees a full

backup of the device, while retaining the storage space advantages of an incremental backup.

One enhancement to physical backup is to copy only blocks that are actively in use in the file system. For most file system formats it is relatively easy to determine if a block is in use or free without walking the file system hierarchy. Free blocks generally contain the remnants of temporary files that were created and removed in the time between backups and it is advantageous not to store such blocks. This optimization requires that the backup format be able to represent missing blocks, which can easily be achieved on Venti by storing a null value for the appropriate entry in the pointer tree.

The random access performance of Venti is sufficiently good that it is possible to use a physical backup without first restoring it to disk. With operating system support, it is feasible to directly mount a backup file system image from Venti. Access to this file system is read only, but it provides a natural method of restoring a subset of files. For situations where a full restore is required, it might be possible to do this restore in a lazy fashion, copying blocks from Venti to the file system as needed, instead of copying the entire contents of the file system before resuming normal operation.

The time to perform a physical backup can be reduced using a variety of incremental techniques. Like vac, the backup utility can compute the fingerprint of each block and compare this fingerprint with the appropriate entry in the pointer tree of a previous backup. This optimization reduces the number of writes to the Venti server. If the file system provides information about which blocks have changed, as is the case with WAFL, the backup utility can avoid even reading the unchanged blocks. Again, a major advantage of using Venti is that the backup utility can implement these incremental techniques while still providing the user with a full backup. The backup utility writes the new blocks to the Venti server and constructs a pointer tree with the appropriate fingerprint for the unchanged blocks.

4.3. Plan 9 File system

When combined with a small amount of read/write storage, Venti can be used as the primary location for data rather than a place to store backups. A new version of the Plan 9 file system, which we are developing, exemplifies this approach.

Previously, the Plan 9 file system was stored on a combination of magnetic disks and a write-once optical jukebox. The jukebox furnishes the permanent storage for the system, while the magnetic disks act as a cache for the jukebox. The cache provides faster file access and, more importantly, accumulates the changes to the file system during the period between snapshots. When a snapshot is taken, new or modified blocks are written from the disk cache to the jukebox.

The disk cache can be smaller than the active file system, needing only to be big enough to contain the daily changes to the file system. However, accesses that miss the cache are significantly slower since changing platters in the jukebox takes several seconds. This performance penalty makes certain operations on old snapshots prohibitively expensive. Also, on the rare occasions when the disk cache has been reinitialized due to corruption, the file server spends several days filling the cache before performance returns to normal.

The new version of the Plan 9 file system uses Venti instead of an optical jukebox as its storage device. Since the performance of Venti is comparable to disk, this substitution equalizes access both to the active and to the archival view of the file system. It also allows the disk cache to be quite small; the cache accumulates changes to the file system between snapshots, but does not speed file access.

5. Implementation

We have implemented a prototype of Venti. The implementation uses an append-only log of data blocks and an index that maps fingerprints to locations in this log. It also includes a number of features that improve robustness and performance. This section gives a brief overview of the implementation. Figure 3 shows a block diagram of the server.

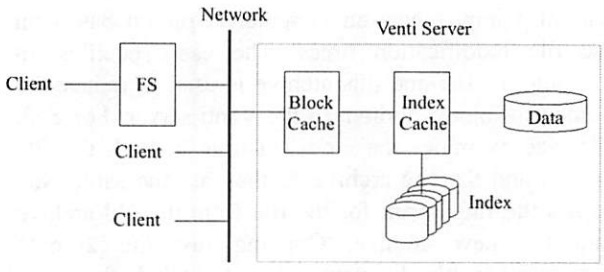


Figure 3. A block diagram of the Venti prototype.

Since Venti is intended for archival storage, one goal of our prototype is robustness. The approach we have

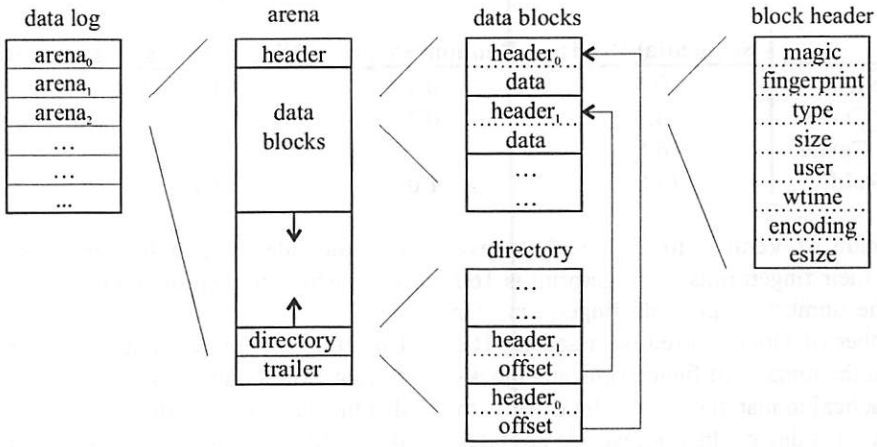


Figure 4. The format of the data log.

taken is to separate the storage of data blocks from the index used to locate a block. In particular, blocks are stored in an append-only log on a RAID array of disk drives. The simplicity of the append-only log structure eliminates many possible software errors that might cause data corruption and facilitates a variety of additional integrity strategies. A separate index structure allows a block to be efficiently located in the log; however, the index can be regenerated from the data log if required and thus does not have the same reliability constraints as the log itself.

The structure of the data log is illustrated in Figure 4. To ease maintenance, the log is divided into self-contained sections called *arenas*. Each arena contains a large number of data blocks and is sized to facilitate operations such as copying to removable media. Within an arena is a section for data bocks that is filled in an append-only manner. In Venti, data blocks are variable sized, up to a current limit of 52 Kbytes, but since blocks are immutable they can be densely packed into an arena without fragmentation.

Each block is prefixed by a header that describes the contents of the block. The primary purpose of the header is to provide integrity checking during normal operation and to assist in data recovery. The header includes a magic number, the fingerprint and size of the block, the time when the block was first written, and identity of the user that wrote it. The header also includes a user-supplied type identifier, which is explained in Section 7. Note, only one copy of a given block is stored in the log, thus the user and wtime fields correspond to the first time the block was stored to the server.

Before storing a block in the log, an attempt is made to compress its contents. The inclusion of data compression increases the effective capacity of the archive and is simple to add given the log structure. Obviously, some blocks are incompressible. The encoding field in the block header indicates whether the data was compressed and, if so, the algorithm used. The esize field indicates the size of the data after compression, enabling the location of the next block in the arena to be determined. The downside of using compression is the computational cost, typically resulting in a decrease in the rate that blocks can be stored and retrieved. Our prototype uses a custom Lempel-Ziv '77 [21] algorithm that is optimized for speed. Compression is not a performance bottleneck for our existing server. Future implementations may benefit from hardware solutions.

In addition to a log of data blocks, an arena includes a header, a directory, and a trailer. The header identifies the arena. The directory contains a copy of the block header and offset for every block in the arena. By replicating the headers of all the blocks in one relatively small part of the arena, the server can rapidly check or rebuild the system's global block index. The directory also facilitates error recovery if part of the arena is destroyed or corrupted. The trailer summarizes the current state of the arena itself, including the number of blocks and the size of the log. Within the arena, the data log and the directory start at opposite ends and grow towards each other. When the arena is filled, it is marked as sealed, and a fingerprint is computed for the contents of the entire arena. Sealed arenas are never modified.

Table 1. The performance of read and write operations in Mbytes/s for 8 Kbyte blocks

	Sequential Reads	Random Reads	Virgin Writes	Duplicate Writes
Uncached	0.9	0.4	3.7	5.6
Index Cache	4.2	0.7	-	6.2
Block Cache	6.8	-	-	6.5
Raw Raid	14.8	1.0	12.4	12.4

The basic operation of Venti is to store and retrieve blocks based on their fingerprints. A fingerprint is 160 bits long, and the number of possible fingerprints far exceeds the number of blocks stored on a server. The disparity between the number of fingerprints and blocks means it is impractical to map the fingerprint directly to a location on a storage device. Instead, we use an index to locate a block within the log.

We implement the index using a disk-resident hash table as illustrated in Figure 5. The index is divided into fixed-sized buckets, each of which is stored as a single disk block. Each bucket contains the index map for a small section of the fingerprint space. A hash function is used to map fingerprints to index buckets in a roughly uniform manner, and then the bucket is examined using binary search. If provisioned with sufficient buckets, the index hash table will be relatively empty and bucket overflows will be extremely rare. If a bucket does overflow, the extra entries are placed in an adjacent bucket. This structure is simple and efficient, requiring one disk access to locate a block in almost all cases.

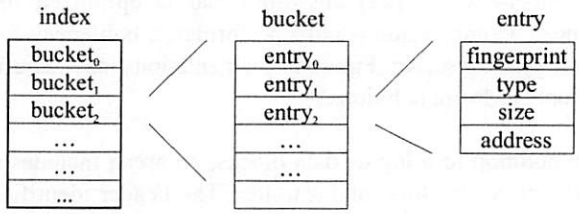


Figure 5. Format of the index.

The need to go through an index is the main performance penalty for Venti compared to a conventional block storage device. Our prototype uses three techniques to increase the performance: caching, striping, and write buffering.

The current implementation has two important caches of approximately equal size: a block cache and an index cache. A hit in the block cache returns the data for that fingerprint, bypassing the both the index lookup and access to the data log. Hits in the index cache eliminate only the index lookup, but the entries are much smaller and the hit rate correspondingly higher.

Unfortunately, these caches do not speed the process of storing a new block to Venti. The server must check that the block is not a duplicate by examining the index. If the block is not contained on the server, it will obviously not be in any cache. Since the fingerprint of the block contains no internal structure, the location of a fingerprint in the index is essentially random. Furthermore, the archival nature of Venti means the entire index will not fit in memory because of the large number of blocks. Combining these factors means that the write performance of Venti will be limited to the random IO performance of the index disk, which for current technology is a few hundred accesses per second. By striping the index across multiple disks, however, we get a linear speedup. This requires a sufficient number of concurrent accesses, which we assure by buffering the writes before accessing the index.

The prototype Venti server is implemented for the Plan 9 operating system in about 10,000 lines of C. The server runs on a dedicated dual 550Mhz Pentium III processor system with 2 Gbyte of memory and is accessed over a 100Mbs Ethernet network. The data log is stored on a 500 Gbyte MaxTronic IDE Raid 5 Array and the index resides on a string of 8 Seagate Cheetah 18XL 9 Gbyte SCSI drives.

6. Performance

Table 1 gives the preliminary performance results for read and write operations in a variety of situations. For comparison, we include the SCSI performance of the RAID array. Although the performance is still several times slower than directly accessing the disk, we believe the results are promising and will improve as the system matures.

The uncached sequential read performance is particularly bad. The problem is that these sequential reads require a random read of the index. Without assistance from the client, the read operations are not

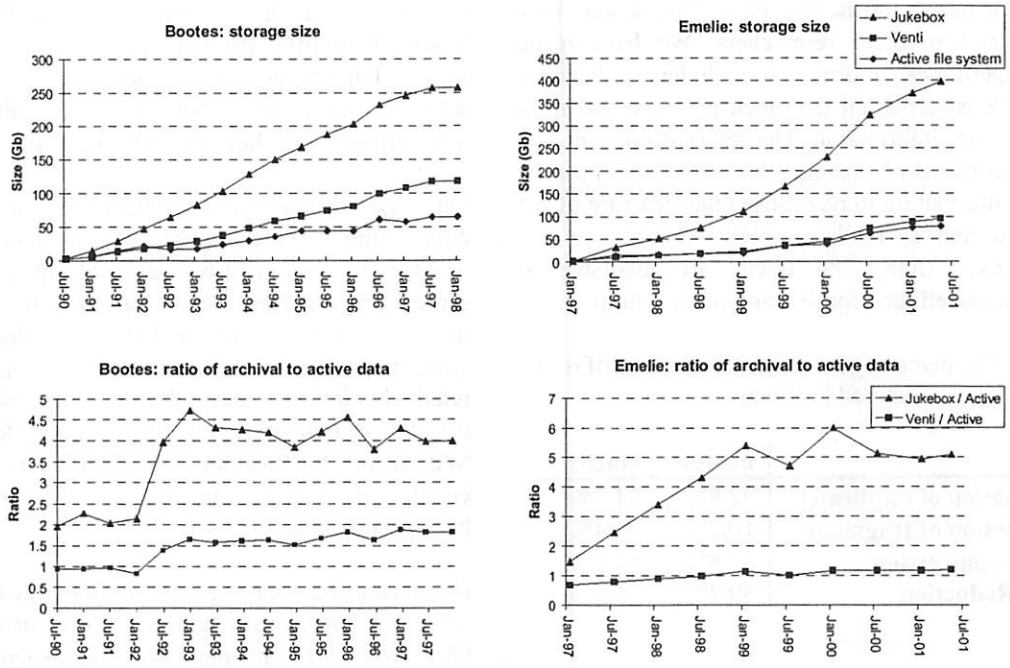


Figure 6. Graphs of the various sizes of two Plan 9 file servers.

overlapped and do not benefit from the striping of the index. One possible solution is a form of read-ahead. When reading a block from the data log, it is feasible to also read several following blocks. These extra blocks can be added to the caches without referencing the index. If blocks are read in the same order they were written to the log, the latency of uncached index lookups will be avoided. This strategy should work well for streaming data such as multimedia files.

The basic assumption in Venti is that the growth in capacity of disks combined with the removal of duplicate blocks and compression of their contents enables a model in which it is not necessary to reclaim space by deleting archival data. To demonstrate why we believe this model is practical, we present some statistics derived from a decade's use of the Plan 9 file system.

The computing environment in which we work includes two Plan 9 file servers named bootes and emelie. Bootes was our primary file repository from 1990 until 1997 at which point it was superseded by emelie. Over the life of these two file servers there have been 522 user accounts of which between 50 and 100 were active at any given time. The file servers have hosted numerous development projects and also contain several large data sets including chess end games, astronomical data, satellite imagery, and multimedia files.

Figure 6 depicts the size of the active file system as measured over time by du, the space consumed on the jukebox, and the size of the jukebox's data if it were to be stored on Venti. The ratio of the size of the archival data and the active file system is also given. As can be seen, even without using Venti, the storage required to implement the daily snapshots in Plan 9 is relatively modest, a result of the block level incremental approach to generating a snapshot. When the archival data is stored to Venti the cost of retaining the snapshots is reduced significantly. In the case of the emelie file system, the size on Venti is only slightly larger than the active file system; the cost of retaining the daily snapshots is almost zero. Note that the amount of storage that Venti uses for the snapshots would be the same even if more conventional methods were used to back up the file system. The Plan 9 approach to snapshots is not a necessity, since Venti will remove duplicate blocks.

When stored on Venti, the size of the jukebox data is reduced by three factors: elimination of duplicate blocks, elimination of block fragmentation, and compression of the block contents. Table 2 presents the percent reduction for each of these factors. Note, bootes uses a 6 Kbyte block size while emelie uses 16 Kbyte, so the effect of removing fragmentation is more significant on emelie.

The 10 year history of the two Plan 9 file servers may be of interest to other researchers. We have made available per-block information including a hash of each block's contents, all the block pointers, and most of the directory information. The traces do not include the actual contents of files nor the file names. There is sufficient information to reconstruct the structure of the file system and to track the daily changes to this structure over time. The traces are available at http://www.cs.bell-labs.com/~seanq/p9trace.html.

Table 2. The percentage reduction in the size of data stored on Venti.

	bootes	emelie
Elimination of duplicates	27.8%	31.3%
Elimination of fragments	10.2%	25.4%
Data Compression	33.8%	54.1%
Total Reduction	59.7%	76.5%

7. Reliability and Recovery

In concert with the development of the Venti prototype, we have built a collection of tools for integrity checking and error recovery. Example uses of these tools include: verifying the structure of an arena, checking there is an index entry for every block in the data log and vice versa, rebuilding the index from the data log, and copying an arena to removable media. These tools directly access the storage devices containing the data log and index and are executed on the server.

The directory structure at the end of each area enhances the efficiency of many integrity and recovery operations, since it is typically two orders of magnitude smaller than the arena, yet contains most of the needed information. The index checking utility, for example, is implemented as a disk based sort of all the arena directories, followed by a comparison between this sorted list and the index. Our prototype currently contains approximately 150 million blocks using 250 Gbytes of storage. An index check takes 2.2 hours, which is significantly less than the 6 hours it takes to read all the log data.

An additional integrity and recovery feature is the association of a type identifier with every block. This 8 bit identifier is included with all client read and write operations and has the effect of partitioning the server into multiple independent domains. The idea is that type indicates the interpretation of the data contained in the block. A client can use this feature, for example, to indicate that a block is the root node for a tree of blocks. Currently, the data format associated with a type is left entirely to the client; the server does not interpret the type other that to use it in conjunction with a fingerprint as the key with which to index a block.

One use of the type identifier is to assist the administrator in locating blocks for which a user has accidentally lost the fingerprint. Using a tool on the server, the data log can be scanned for blocks that match specified criteria, including the block type, the write time, and user identifier. The type makes it relatively simple to locate forgotten root blocks. Future uses for the type might include the ability for the server to determine the location of fingerprints within a block, enabling the server to traverse the data structures that have been stored.

By storing the data log on a RAID 5 disk array, our server is protected against single drive failures. Obviously, there are many scenarios where this is not sufficient: multiple drives may fail, there may be a fire in the machine room, the RAID firmware may contain bugs, or the device may be stolen.

Additional protection could be obtained by using one or more off-site mirrors for the server. We have not yet implemented this strategy, but the architecture of Venti makes this relatively simple. A background process on the server copies new blocks from the data log to the mirrors. This copying can be achieved using the Venti protocol; the server is simply another client to the mirror.

Even mirroring may not be sufficient. The implementation of Venti may contain bugs that can be exploited to compromise the server. An automated attack may delete data on many servers simultaneously. Storage devices that provide low level enforcement of a write-once policy would provide protection for such an attack. Write-once read-many optical jukeboxes often provide such protection, but this is not yet common for magnetic disk based storage systems. We have thus resorted to copying the sealed arenas onto removable media.

8. Related Work

The Stanford Archival Vault [2] is a prototype archival repository intended for digital libraries. The archive consists of a write-once log of digital objects (files) and several auxiliary indexes for locating objects within the log. Objects are identified by the hash of their contents using a cyclic redundancy check (CRC). Unlike Venti,

this system has no way to share data between objects that are partially the same, or to build up complex data structures such as a file system hierarchy. Rather, the archive consists of a collection of separate objects with a limited ability to group objects into sets.

On Venti, blocks are organized into more complex data structures by creating hash-trees, an idea originally proposed by Merkle [11] for an efficient digital signature scheme.

The approach to block retrieval in the Read-Only Secure File System (SFSRO) [3] is comparable to Venti. Blocks are identified by the Sha1 hash of their contents and this idea is applied recursively to build up more complex structures. The focus of this system is security, not archival storage. An administrator creates a digitally signed database offline. The database contains a public read-only file system that can be published on multiple servers and efficiently and securely accessed by clients. SFSRO outperforms traditional methods for providing data integrity between a client and a file server, demonstrating an attractive property of hash-based addressing.

Given their similarities, it would be simple to implement SFSRO on top of Venti. The goal of Venti is to provide a flexible location for archival storage and SFSRO is a good example of an application that could use this capability. In fact, using Venti would provide a trivial solution to SFSRO's problem with stale NFS handles since data is never deleted from Venti and thus a stale handle will never be encountered.

Content-Derived Names [6] are another example of naming objects based on a secure hash of its contents. This work addresses the issue of naming and managing the various binary software components, in particular shared libraries, that make up an application.

The philosophy of the Elephant file system [18] is similar to Venti; large, cheap disks make it feasible to retain many versions of data. A feature of the Elephant system is the ability to specify a variety of data retention policies, which can be applied to individual files or directories. These policies attempt to strike a balance between the costs and benefits of storing every version of a file. In contrast, Venti focuses on the problem of how to store information after deciding that it should be retained in perpetuity. A system such as the Elephant file system could incorporate Venti as the storage device for the permanent "landmark" versions of files, much as the Plan 9 file system will use Venti to archive snapshots.

Self-Securing Storage [19] retains all versions of file system data in order to provide diagnosis and recovery from security breaches. The system is implemented as a self-contained network service that exports an object-based disk interface, providing protection from compromise of the client operating system. Old data is retained for a window of time and then deleted to reclaim storage.

Venti provides many of the features of self-securing storage: the server is self-contained and accessed through a simple low-level protocol, malicious users cannot corrupt or delete existing data on the server, and old versions of data are available for inspection. It is unlikely that a system would write every file system operation to Venti since storage is never reclaimed, but not deleting data removes the constraint that an intrusion must be detected within a limited window of time. A hybrid approach might retain every version for some time and some versions for all time. Venti could provide the long-term storage for such a hybrid.

9. Future Work

Venti could be distributed across multiple machines; the approach of identifying data by a hash of its contents simplifies such an extension. For example, the IO performance could be improved by replicating the server and using a simple load balancing algorithm. When storing or retrieving a block, clients direct the operation to a server based on a few bits of the fingerprint. Such load balancing could even be hidden from the client application by interposing a proxy server that performs this operation on behalf of the client.

Today, Venti provides little security. After authenticating to the server, clients can read any block for which they know the fingerprint. A fingerprint does act as a capability since the space of fingerprints is large and the Venti protocol does not include a means of enumerating the blocks on the server. However, this protection is weak as a single root fingerprint enables access to an entire file tree and once a fingerprint is known, there is no way to restrict access to a particular user. We are exploring ways of providing better access control.

To date, the structures we have used for storing data on Venti break files into a series of fixed sized blocks. Identical blocks are consolidated on Venti, but this consolidation will not occur if the data is shifted within the file or an application uses a different block size. This limitation can be overcome using an adaptation of

Manber's algorithm for finding similarities in files [9]. The idea is to break files into variable sized blocks based on the identification of anchor or break points, increasing the occurrence of duplicate blocks [12]. Such a strategy can be implemented in client applications with no change to the Venti server.

A more detailed analysis of the decade of daily snapshots of the Plan 9 file systems might be interesting. The trace data we have made publicly available contains approximately the same information used for other studies of long term file activity [4].

10. Conclusion

The approach of identifying a block by the Sha1 hash of its contents is well suited to archival storage. The write-once model and the ability to coalesce duplicate copies of a block makes Venti a useful building block for a number of interesting storage applications.

The large capacity of magnetic disks allows archival data to be retained and available on-line with performance that is comparable to conventional disks. Stored on our prototype server is over a decade of daily snapshots of two major departmental file servers. These snapshots are stored in a little over 200 Gbytes of disk space. Today, 100 Gbytes drives cost less than $300 and IDE RAID controllers are included on many motherboards. A scaled down version of our server could provide archival storage for a home user at an attractive price. Tomorrow, when terabyte disks can be had for the same price, it seems unlikely that archival data will be deleted to reclaim space. Venti provides an attractive approach to storing that data.

11. Acknowledgments

This paper was improved by comments and suggestions from Peter Bosch, Eric Grosse, Lorenz Huelsbergen, Rob Pike, Ross Quinlan, and Cliff Young and six anonymous reviewers. The paper's shepherd was Ethan L. Miller. We thank them all for their help.

12. References

[1] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings Joint NASA and IEEE Mass Storage Conference*, March 1998.

[2] Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the 3rd ACM International Conference on Digital Libraries*, 1998.

[3] Kevin Fu, Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[4] Timothy J. Gibson, Ethan L. Miller, and Darrell D. E. Long. Long-term file activity and inter-reference patterns. In *Proceedings, 24th International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems, Computer Measurement Group*, December 1998.

[5] Dave Hitz, James Lau, and Michael Malcolm, File system design for an NFS file server appliance, In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, January 1994.

[6] J. K. Hollingsworth and E. L. Miller. Using content-derived names for configuration management. In *Proceeding of the 1997 ACM Symposium on Software Reusability*, Boston, May 1997.

[7] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.

[8] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[9] Udi Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, January 1994.

[10] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[11] Ralph C. Merkle. Protocols for public-key cryptosystems. In *Proceedings of the IEEE*

Symposium on Security and Privacy, pp. 122-133, April 1980.

[12] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th Symposium on Operating Systems Principles*, October 2001.

[13] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. US Department of Commerce, April 1995.

[14] National Institute of Standards and Technology, Draft FIPS 180-2. Secure Hash Standard. US Department of Commerce, May 2001.

[15] Evi Nemeth, Garth Snyder, Scott Seebass, and Trent R. Hein. *UNIX System Administration Handbook 3rd Edition*, Prentice Hall, 2001.

[16] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs, *Computing Systems*, Vol. 8, 3, pp. 221-254, Summer 1995.

[17] Sean Quinlan. A cache worm file system. *Software-Practice and Experience*, Vol 21, 12, pp 1289-1299, December 1991.

[18] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th Symposium on Operating Systems Principles*, December 12-15, 1999.

[19] John. D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[20] D. A. Thompson and J. S. Best. The future of magnetic data storage technology, *IBM Journal of Research and Development*, Vol 44, 3, pp. 311-322, May 2000.

[21] J. Ziv and A. Lempel. A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337-343, May 1977.

Myriad: Cost-effective Disaster Tolerance

Fay Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon E. Perl, Li Zhang
Compaq Systems Research Center
Palo Alto, CA, 94301

Abstract

This paper proposes a new approach for achieving disaster tolerance in large, geographically-distributed storage systems. The system, called *Myriad*, can achieve the same level of disaster tolerance as a typical single mirrored solution, but uses considerably fewer physical resources, by employing cross-site checksums (via erasure codes) instead of direct replication.

The key technical contribution of the paper is a protocol permitting cross-site checksums to be updated in such a way that data recovery is always possible. Another important contribution is the specification of a protocol for recovering from disasters, explicitly verifying the claim of disaster tolerance. Further, it is shown by direct calculation and analytical modeling that Myriad compares favorably with mirroring in terms of both total cost of ownership and reliability.

1 Introduction

A *geoplex* is a collection of geographically distributed sites, each consisting of servers, applications, and data [7]. The sites of a geoplex cooperate to improve reliability and/or availability of applications and data through the use of redundancy. Data redundancy in geoplexes typically takes the form of *mirroring*, where one or more full copies of the logical data are maintained at remote sites. In this paper we present alternative approaches to mirroring for cross-site data redundancy in geoplexes. While the alternatives are not as generally applicable as mirroring, they have noticeably lower cost, provide additional flexibility, and are appropriate for a significant class of applications.

Mirroring has a number of desirable properties. It is conceptually simple, and does not compromise performance when it operates in an asynchronous mode for remote updates. Its recovery procedure is simple. In addition to the ability to reconstruct data after a site failure, it offers the choice of active-active configurations (where all sites are actively processing some work) and active-passive configurations (where fast failover is possible from the primary site to a secondary site). On the negative side, mirroring has a high cost because the amount of storage required for the logical data must be doubled or more depending on the number of mirror copies. For very high reliability, more than one mirror copy generally is required. While the high cost for remote mirroring has been accepted by customers with mission-critical applications, such as online transaction processing systems, a geoplex is hardly a low-cost product available for many other applications with large data sets, such as data mining and scientific computing.

We investigate the possibility of offering more flexible, often lower, pricing of a geoplex by supporting a variety of data redundancy schemes across sites. The system, called Myriad, uses redundancy schemes based on *erasure codes* [16] (error-correcting codes where the position of the error is known). Erasure codes form the basis for the approaches to disk array redundancy known as RAID levels 3, 4, 5, and 6 [10]. We study this approach for maintaining redundancy of data spread across geographically distant sites rather than across disks in a disk array.

We start by examining the reasons erasure codes have not previously been employed in geoplexes. It is perceived that although erasure codes reduces the number of disks needed for a given amount of data, the dollar saving is insignificant because disks account for only a small portion (10-25%) of the total cost of ownership (TCO) of the entire storage solution. There is a perception that the software implementation of erasure codes across servers offers lower reliability than mirroring, and it is too complicated to be commercializable even within a local site.

The TCO includes all costs attributable to a storage system over its lifetime, including purchase, installation, power, floor space and human labor costs for administering and maintaining the system. By analyzing components of the TCO, we discover that requiring less hardware lowers not just the purchase cost but also other TCO components such as environmental and administration costs. Therefore, a scheme using less hardware could reduce the TCO by a noticeable amount. For example, our analysis shows that, in order to implement a geo-

Sharon E. Perl and Shun-Tak A. Leung are currently with Google, Inc. and can be reached at {sharon, shuntak}@google.com. The other authors can be reached at {Fay.Chang, Minwen.Ji, John.MacCormick, L.Zhang}@compaq.com.

plex across 5 sites with 20 terabytes of data on each site, single mirroring costs about 80% more than the original (non-redundant) scheme, while a parity-based scheme (an instance of an erasure code) costs only about 40% more. Section 3 presents our TCO analysis.

We also compare the reliability of various cross-site redundancy schemes, primarily through analytical modeling. We derive equations for the mean time to data loss (MTTDL) of these schemes. Across a spectrum of system configurations, we find, not surprisingly, that the MTTDL of a Myriad scheme with one checksum (single redundancy) is worse than that of a mirrored system but much better than if there is no cross-site redundancy. Moreover, the MTTDL of a Myriad scheme with two checksums (double redundancy) is worse than that of a double mirroring system but much better than that for a mirrored system. Section 4 discusses the results of the reliability analysis in more details.

The complexity of non-mirroring redundancy schemes is an issue for both local-area and cross-site storage systems. However, we have observed enough differences in the two systems to believe that their protocols should be quite different. In a local-area parity scheme, such as the storage layer in xFS [20], the local-area network (LAN) connecting servers is assumed to be fast and cheap (e.g., Ethernet). The design goal is to parallelize reads and writes across disks for large aggregate bandwidth and to present an image of a single system. Therefore, the design challenges lie in data layout, coordination across servers, cache coherence, and decentralization of control.

In the systems we are targeting, the wide-area network (WAN) connecting sites is assumed to be slow and expensive (e.g., leased T1 or T3 lines). Consequently, applications running on each site are configured to be independent of applications running on other sites; in other words, a logical piece of data is not stored across sites. For example, suppose a hospital chain has branches in multiple cities and each branch has its own file system for local employees and patients. In order to implement Myriad-style data redundancy across branches, one only needs to add a certain amount of physical storage to each branch, which will be dedicated to storing the redundancy information (i.e., checksums) of data on other sites. Although branches may manage to gain access to each other by mounting remote file systems, the storage layout of the local file systems does not have to be changed. Therefore, at the block storage level, there is no issue about parallelism or single-system image across sites. Rather, the goal of our design is to reliably and consistently deliver data to the remote checksum sites for protection while hiding the long latency of WAN from the critical path of data access.

We have designed a cross-site update/recovery protocol that supports redundancy based on erasure codes, where the number of data and checksum disks may vary. As it happens, mirroring is a degenerate case of erasure coding where there is a single data block per group of checksum blocks, so our protocol supports mirroring as well. However, it is expected to have higher complexity and overhead than a protocol designed solely for mirroring. There are two major reasons for this: (1) the reconstruction of data with a non-mirroring scheme is substantially more difficult than with a mirroring scheme; (2) the operation of updating a checksum is not idempotent, in contrast to that of a mirror. Nevertheless, we design our protocol so that it requires no more WAN bandwidth than a pure mirroring protocol. Sections 5 and 6 discuss our design in detail.

2 System Overview

A Myriad system achieves disaster tolerance by storing data at a number of geographically distinct sites. Each site consists of disks, servers, a LAN, and some local redundancy such as hardware RAID-5; the sites are connected by a WAN. Each site is assumed to employ a storage area network (SAN).

The essential idea behind Myriad is that, in addition to any local redundancy such as RAID, each block of data participates in precisely one *cross-site redundancy group*. A cross-site redundancy group is a set of blocks, one per site, of which one or more are checksum blocks. Thus, the blocks in a given group protect one another's data. The simplest possible example is single parity, in which one of the blocks is the XOR of the others — this is equivalent to running a distributed form of RAID-5. The system can reconstruct the current, correct contents of a lost site on a replacement site.

Much greater disaster tolerance can be achieved by using more redundancy. For instance, one can use all but two of the blocks in every cross-site redundancy group for data, and use the remaining two blocks as checksums computed using a Reed-Solomon erasure code [2]. This type of the system, which is equivalent to running distributed RAID-6, can recover from up to two site losses.

An application using a Myriad storage system must satisfy two properties:

1. **Dispersed data**: data is dispersed over multiple sites. The Myriad protocol (section 6) formally requires as few as two sites (single redundancy) or three sites (double redundancy), but as discussed in section 3, the efficiency gains of the Myriad system (as compared with mirroring) are more compelling when there are more (say 5 or more). In addition, the amount of data at each site should be roughly equal; otherwise, the efficiency gains are again reduced. (This is the same as the problem of using disks of different sizes in a RAID-5 array.)

2. **Local computation**: computations on the data are collocated with the data. In other words, an application running at a given site does not access data at other sites. This assumption is motivated by the economic justification of the Myriad approach: if computations are not local, the cost of WAN bandwidth is likely to exceed Myriad's cost benefits which result from using less physical storage.

A relatively broad class of storage customers meet both the "dispersed data" and the "local computation" conditions. A typical such customer has several different sites, each of which runs its own application and storage system. For example, different sites might perform various payroll, finance, and technical functions. Alternatively, as with the example of a hospital chain given earlier, the sites could be running independent instantiations of the same application, but using only local data. Another potential customer for Myriad is an application service provider (ASP) or storage service provider (SSP) that wants to offer disaster tolerance to their customers cost-effectively.

3 Total Cost of Ownership

The present paradigm for achieving disaster tolerance in a storage system is to mirror all data at a remote site. However, mirroring is expensive: the amount of physical data is twice the amount of logical data (often referred to as "100% space overhead"), so one must purchase and administer twice as much storage as for a basic (disaster-vulnerable) system. As described later, a typical Myriad system with 5 sites might have only 20–40% space overhead, while retaining or even improving on the disaster tolerance of a mirrored system. So the physical requirement and hence purchase cost for Myriad are as much as 40% ($= 1 - \frac{100+20}{100+100}$) less than those for mirroring. Although the purchase cost of a storage system is widely known to be only a small fraction of the TCO (reports estimate 10–25% [1, 12, 18]), this section will show by explicit calculation that a Myriad system would still represent significant TCO savings over a mirrored system.

3.1 Cost Model and Assumptions

A good starting point for calculating the TCO of the system is a report by Gartner Group [1], which estimates the components for the storage TCO of a single-site system; the estimates are shown in Figure 1. In the following analysis, the Hardware Management category is combined with Administration, and the Downtime category is eliminated since it is an opportunity cost related to system reliability.

We first determine what proportion of each cost category scales with the physical, as opposed to logical,

cost category	% of storage TCO
administration	13%
purchase	20%
environmentals	14%
backup/restore	30%
hardware management	3%
downtime	20%

Figure 1: Storage TCO (Source: Gartner, "Don't Waste Your Storage Dollars: What you Need To Know", Nick Allen, March 2001 [1].)

amount of storage. Specifically, let Physical be the amount of physical storage, Logical the amount of logical storage (i.e., storage for user data, including local redundancy), and C_{admin}, C_{purch}, C_{env} and C_{backup} the administration, purchase, environmental and backup/restore costs respectively. Each type of cost is modeled as a linear combination of Physical and Logical. For example,

$$C_{\text{admin}} = \alpha_{\text{admin}}(\lambda_{\text{admin}}\text{Physical} + (1 - \lambda_{\text{admin}})\text{Logical}) \quad (1)$$

Intuitively, each category is parameterized by $\lambda \in [0, 1]$ specifying how much the cost depends on Physical rather than Logical. This defines an "effective storage size" λPhysical$+(1-\lambda)$Logical for the category. The absolute cost of the category is obtained by multiplying by a coefficient α, which is the category cost in \$/GB of effective storage.

Appropriate values for the λ and α parameters can be inferred as follows. First, $\lambda_{\text{purch}} = 1$ by definition. Environmental costs include power, UPS, and floor space, all of which scale directly with Physical, so $\lambda_{\text{env}} = 1$ also. The value of α_{purch} can be determined directly from published component prices (see Figure 2 for an example), and so we express the remaining α-values in terms of α_{purch}. Following the proportions in Figure 1, we take $\alpha_{\text{admin}} = \frac{16}{20}\alpha_{\text{purch}}$, $\alpha_{\text{env}} = \frac{14}{20}\alpha_{\text{purch}}$, $\alpha_{\text{backup}} = \frac{30}{20}\alpha_{\text{purch}}$. That leaves λ_{backup} and λ_{admin}. We estimate λ_{backup} to be 0. This is conservative in that that it makes a mirrored solution look as good as possible in comparison to Myriad. As for λ_{admin}, by itemizing the tasks of a system administrator and considering whether each depends primarily on physical or logical data size, we estimate λ_{admin} to be 0.5. (Tasks scaling primarily with logical data size include most software management tasks, such as array control management, cross-site redundancy management, snapshot operation, and local network management. Tasks scaling primarily with physical data size include monitoring, reporting on, and altering physical storage resources, and implementing volume growth.) But since the value of λ_{admin} may be controversial, we leave it as a free variable for now.

item	no.	price ($)	cost ($K)	component
NICs	46	600	28	NC6134 GB NIC 1Gbps
enclosures	46	3500	161	StorageWorks 4354R
drives	616	900	580	36.4GB 10K Ultra3
LAN port	46	400	18	Asante IntraCore 65120
controllers	46	800	37	Smart Array 431
total			823	

Figure 2: Purchase cost breakdown for a typical 20TB storage system, based on component prices posted on Compaq and Asante web sites.

Let Overhead be the space overhead of the remote redundancy scheme, so that Physical $=$ (1 + Overhead) $\times$ Logical, and let C_{WAN} denote the cost of WAN bandwidth consumed by the system over its lifetime. After substitutions and simplifications, we arrive at

$$\text{TCO} = \alpha_{\text{purch}}\,\text{Logical}\,[(0.8\lambda_{\text{admin}} + 1.7) \times \text{Overhead} + 4.0] + C_{\text{WAN}}. \quad (2)$$

For concreteness, consider a storage system with 100TB of logical data distributed over five sites (or 20TB/site). If each site runs RAID-5 locally in hardware, and reserves one hot spare in every 14-drive enclosure, the purchase cost of physical equipment is about \$823K/site for the particular choice of components listed in Figure 2; this corresponds to a value of $\alpha_{\text{purch}} =$ \$42/GB.

To calculate the lifetime WAN cost C_{WAN}, note that WAN bandwidth is only for redundancy information updates because client data accesses are local. Assume that bandwidth costs \$500/Mbps/month/site, that the lifetime of the system is five years, that all the data is overwritten on average twice per year, and that the system achieves an average 33% utilization of the purchased bandwidth, due to burstiness. (The bandwidth cost is typical of ISPs at the time of writing; the other numbers are just examples chosen here for concreteness — the actual data write rate and burstiness are highly application-dependent.) This gives

$$C_{\text{WAN}} = \frac{\text{Logical} \times \text{bandwidth cost} \times \text{lifetime}}{\text{utilization} \times \text{turnover period}} = \$4.8\text{M/year}$$

The resulting bandwidth requirement between any two sites is 16Mb/s. The bandwidth costs are doubled for double redundancy, whether in a mirroring scheme or in Myriad. Additional bandwidth is required for recovery; this adds less than 1% to the WAN cost using worst-case parameters from the next section, assuming that the price for extra bandwidth is the same as for standard bandwidth. Although ISPs do not currently sell bandwidth in this "expandable" manner, this may change in the near future [8].

3.2 Results and Discussion

Figure 3(a) shows the TCO for such a system with varying values for the space overhead of the remote redundancy scheme, assuming five sites and $\lambda_{\text{admin}} = 0.5$. Note that Myriad with one checksum site (remote RAID-5, Overhead $= 25\%$) costs 22% less than a standard singly-mirrored system (Overhead $= 100\%$), and Myriad with two checksum sites (remote RAID-6, Overhead $= 67\%$) costs 27% less than double-mirroring (Overhead $= 200\%$). Figure 3(b) shows a breakdown into cost categories.

Our sensitivity analysis finds that these results are not too sensitive to λ_{admin}. They are somewhat sensitive, however, to the assumptions about WAN bandwidth requirements. If the between-site requirement is in fact half the earlier estimate (i.e., 8Mb/s), Myriad's cost advantages are 24% and 30%. If the requirement is twice that (i.e., 32Mb/s), they drop to 19% and 22%. Also note that the cost of a Myriad system with two checksum sites is only 6% above the cost of a standard singly-mirrored system. As Section 4 shows, this small additional cost buys significantly more reliability.

Of course, the model (1) is not sufficiently realistic to predict the costs of all storage systems — such systems vary too widely for any single formula to be accurate. Nevertheless, we believe this model conveys the essence of how TCO depends on physical data size, and hence yields a valid comparison between the mirroring and Myriad approaches.

Finally, the above analysis assumes that the raw data cannot be significantly compressed (perhaps because it is already compressed). Otherwise, mirrored systems could become more attractive by compressing their remote copy.

4 Reliability

The previous section argued that, contrary to common supposition, the cost of disaster tolerance is highly dependent on the storage overhead of the cross-site data redundancy scheme. In this section, we study the reliability of different cross-site data redundancy schemes and demonstrate that lower-overhead schemes can provide substantial reliability benefits.

We use the mean time to data loss (MTTDL) as our reliability metric. In analyzing the MTTDL of a multi-site storage system, we assume that each site is already using some local data redundancy scheme. In particular, we assume that the blocks at each site are stored on hardware RAID boxes that are implementing a RAID level

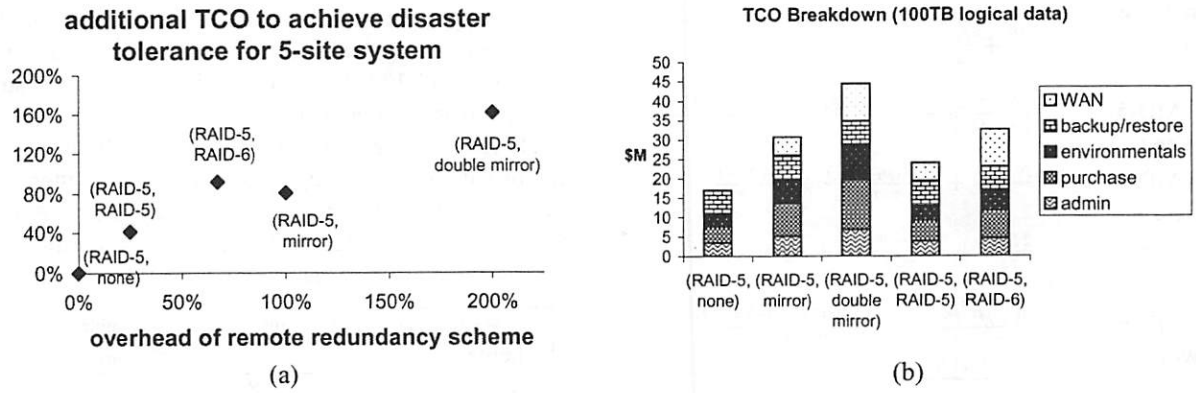


Figure 3: (a) The TCO given by equation (2). The vertical axis is the cost of adding disaster-tolerance to a raw (i.e. non-disaster-tolerant) system, expressed as a percentage of the cost of a raw system. Space overhead Overhead is on the horizontal axis, and $\lambda_{\text{admin}} = 0.5$. Five specific combinations of redundancy schemes are marked, assuming five sites. The key is (local redundancy scheme, remote redundancy scheme). Note that the Myriad systems — (RAID-5, RAID-5) and (RAID-5, RAID-6) — deliver significant savings over the corresponding mirrored solutions, respectively (RAID-5, mirror) and (RAID-5, double mirror). (b) Breakdown of the TCO, using the same assumptions as (a), and $\alpha_{\text{purch}} = \$42/\text{GB}$, Logical = 100TB.

that provides some redundancy (i.e., any level other than RAID-0). This is a reasonable assumption because hardware RAID boxes enable fast redundant updates, and both good performance during and fast recovery from the common failure cases. We also make the simplifying assumption that only two types of components need to be considered in our reliability analysis — hardware RAID boxes and sites — and that failures of different components are exponentially distributed [10], independent, and complete (i.e. when a component fails, all blocks on that component are lost). Note that the analysis is a comparison of hardware failures only; other types of failure (such as software and operator errors) can be significant sources of data loss, but they are not addressed here.

Consider a storage system with D data (as opposed to checksum) blocks spread out over N sites. Assume that each hardware RAID box contains approximately B raw blocks not used for the local parity scheme, such that there are $R = D/B$ RAIDs-worth of data in the system. Let τ_r and τ_s represent the mean time to failure of a hardware RAID box and a site, respectively, and ρ_r and ρ_s represent the mean time to repair of a hardware RAID box and site, respectively. Finally, let $\binom{n}{r}$ be the standard binomial coefficient.

We calculate the MTTDL of the system with five different cross-site redundancy schemes: no cross-site redundancy, cross-site RAID-5, cross-site mirroring, cross-site RAID-6, or cross-site double mirroring. It's important to realize that MTTDL for cross-site RAID depends on the precise layout of redundancy groups. Take cross-site RAID-5 as an example: if the "parity partners" of blocks in a given physical RAID box are distributed randomly across boxes at other sites, any pair of box failures at distinct sites causes data loss. We call this the "worst-case layout", and include its results primarily for completeness. An implementation would certainly avoid this worst case, and strive to achieve the "best-case layout", in which blocks from any physical RAID box partner with blocks from only one box at each other site: with this layout, fewer pairs of failures lead to data loss, and the MTTDL is larger. It's easy to achieve this best case for mirroring, so only the best-case results are shown for the mirroring schemes.

Under these assumptions, standard manipulations of the exponential distribution [10, 19] lead to the formulas for 1/MTTDL shown in Figure 4. To obtain good performance, it is assumed that write calls to the storage system are permitted to return as soon as the write has been committed in a locally redundant fashion. Therefore, MTTDL should also include terms for failures of RAID boxes which have data that has been committed locally but not remotely. However, the amount of "remotely uncommitted" data can be traded off with local write performance in a manner analogous to the trade-off between locally unprotected data and write performance in an AFRAID system [17]. If one assumes the network is reliable, such data loss can be made vanishingly small, and accordingly we neglect it here. The investigation of unreliable networks and details of the write performance-data loss trade-off are left to future work.

Figure 5 shows MTTDL of five-site storage systems, calculated using the equations above. Each graph shows

cross-site scheme	1/MTTDL
None	$\frac{R}{\tau_r} + \frac{N}{\tau_s}$
RAID-5 Worst	$\frac{NR^2\rho_r}{(N-1)\tau_r^2} + \frac{NR(\rho_r+\rho_s)}{\tau_r\tau_s} + \frac{N(N-1)\rho_s}{\tau_s^2}$
RAID-5 Best	$\frac{NR\rho_r}{\tau_r^2} + \frac{NR(\rho_r+\rho_s)}{\tau_r\tau_s} + \frac{N(N-1)\rho_s}{\tau_s^2}$
Mirror	$\frac{2R\rho_r}{\tau_r^2} + \frac{2R(\rho_r+\rho_s)}{\tau_r\tau_s} + \frac{N\rho_s}{\tau_s^2}$
RAID-6 Worst	$\frac{3\binom{N}{3}R^3\rho_r^2}{(N-2)^3\tau_r^3} + \frac{\binom{N}{2}R^2\rho_r(\rho_r+2\rho_s)}{(N-2)\tau_r^2\tau_s} + \frac{\binom{N}{2}R\rho_s(2\rho_r+\rho_s)}{\tau_r\tau_s^2} + \frac{3\binom{N}{3}\rho_s^2}{\tau_s^3}$
RAID-6 Best	$\frac{\binom{N}{2}R\rho_r^2}{\tau_r^3} + \frac{\binom{N}{2}R\rho_r(\rho_r+2\rho_s)}{\tau_r^2\tau_s} + \frac{\binom{N}{2}R\rho_s(2\rho_r+\rho_s)}{\tau_r\tau_s^2} + \frac{3\binom{N}{3}\rho_s^2}{\tau_s^3}$
Double Mirror	$\frac{3R\rho_r^2}{\tau_r^3} + \frac{3R\rho_r(\rho_r+2\rho_s)}{\tau_r^2\tau_s} + \frac{3R\rho_s(2\rho_r+\rho_s)}{\tau_r\tau_s^2} + \frac{N\rho_s^2}{\tau_s^3}$

Figure 4: Formulas for 1/MTTDL for various redundancy schemes

MTTDL for storage systems that contain some particular amount of data, such that results are shown for a wide range of potential storage system sizes (approximately 10 TB to 1 PB, assuming the components listed in Figure 2). Reliability is calculated using a set of conservative failure parameters, and a set of more optimistic failure parameters. For example, our conservative τ_r (RAID box MTTDL) is 150 years, which is less than the MTTF of a single disk drive. Note that, in contrast to some previous work [10, 17], our τ_r includes only failures that cause data loss. For example, we are not including the common case of a RAID controller failure after which all the data can be retrieved simply by moving the disks to another RAID box. NVRAM failures were also neglected.

We also developed an event-driven simulator to investigate factors that were difficult to include in the analytical model. In particular, we investigated whether temporary outages (of sites and hardware RAID boxes) or WAN bandwidth limitations substantially changed the results shown above. We found that both outages and WAN bandwidth limitations did shift the curves, but the shifts were small and did not change our conclusions. The intuition behind why the shifts were small is that, in both cases, the effect is essentially the same as slightly increasing the mean times to recovery, ρ_s and ρ_r.

The most important observation to be made from Figure 5 is that if blocks are distributed according to a "best-case layout", the MTTDL of a RAID-5 system is 2–3 times worse than that of a mirroring system, but much better (around 100 times better) than that of a system with no cross-site redundancy. Furthermore, the MTTDL of a RAID-6 system is worse (by a factor of 10 or so) than that of a double mirroring system, but much better (50–1000 times better) than for a mirrored system. A final summary of the TCO and reliability analyses, using only the base case numbers, is:

cross-site scheme	none	RAID-5	mirror	RAID-6	double mirror
cost	1	1.4	1.8	1.9	2.6
MTTDL	1	100	300	10^5	10^6

where all the numbers are multipliers based on the index "none" = 1.

5 Cross-Site Redundancy

We now go on to describe our design for cross-site redundancy based on erasure codes. This section gives an overview of how we maintain redundancy information, including in particular a static scheme for grouping blocks across sites. The next section describes our protocol for update and recovery.

In a bird's eye view of Myriad, local storage systems on different sites, each serving only local clients, cooperate to achieve disaster tolerance for client data. On each site, the local storage system provides a *logical disk* abstraction to its clients. Clients see only a logical address space divided into blocks of some fixed size, which we call *logical blocks*. Each logical block is identified by its *logical address*. Clients read or write logical blocks; the storage system manages physical data placement. Such a storage system in itself poses many important design issues, but they are beyond the scope of this paper. Petal [14] is one such system.

Local storage systems on different sites cooperate to protect data against site disasters by forming *cross-site redundancy groups*. A cross-site redundancy group (or "group" for short) consists of logical blocks (which we call *data blocks* since they contain client data), and *checksum blocks*, which contain checksums computed from the data blocks. The data and checksum blocks in a group come from different sites, which we call the *data sites* and *checksum sites* of the group. Each group is globally identified by a *group id*, and each data block by its site and (site-specific) logical address.

To tolerate at most m simultaneous site disasters, each group should consist of n ($n > 1$) data blocks and m ($m > 1$) checksum blocks for a geoplex with $n + m$ sites. As for the encoding of the checksum, similar to previous approaches (e.g. [2]), we use a Reed-Solomon

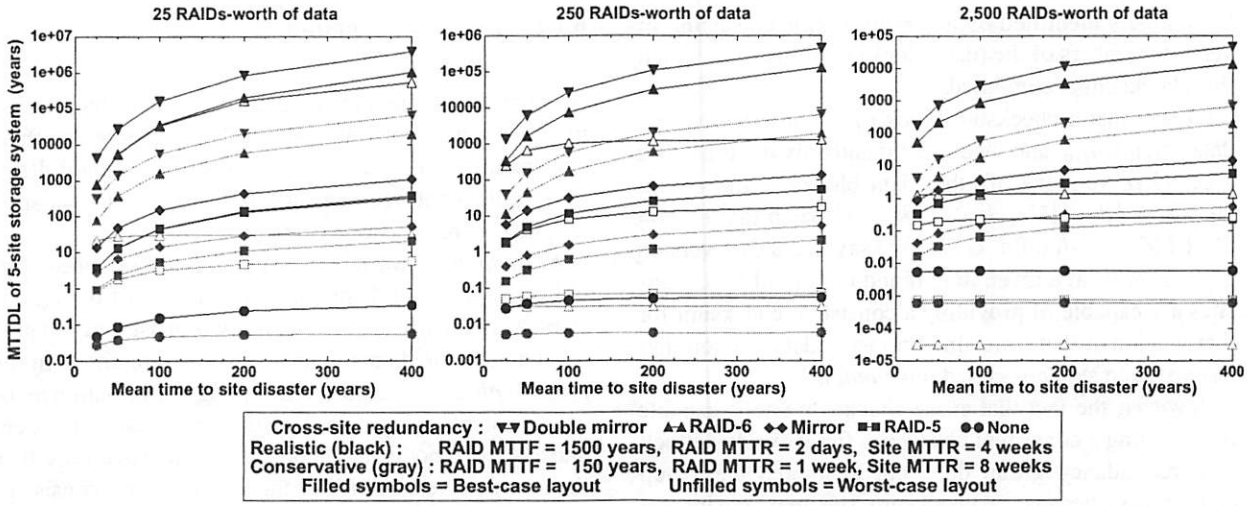


Figure 5: Mean time to data loss for different cross-site redundancy schemes. The results on each graph are for systems with the same amount of logical data D, but different amounts B of data per RAID box, and hence differing values of $R = D/B$, the total number of RAID boxes. Left: $R = 25$. Middle: $R = 250$. Right: $R = 2500$.

code ([16] Ch. 9) to allow incremental update. In other words, a checksum site can compute the new checksum using the old checksum and the XOR between the old and new contents of the data block updated, instead of computing the new checksum from scratch. It degenerates to parity for $m = 1$.

Blocks can be grouped in various ways. In our scheme, we require the checksum blocks be distributed so that they rotate among the sites for logically consecutive data blocks at each individual site. This can be done by using some simple static function to map each data block to a group number and a site number. The following is the function we use. The checksum sites for group g are sites $(g - j) \mod N$, where $0 \leq j < m$. The bth data block at site s is mapped into group g, where

$$g = \begin{cases} b + m \cdot (\lfloor (b-s)/n \rfloor + 1) & s \leq n, \\ b + s - n + m \cdot \lfloor b/n \rfloor & s > n. \end{cases}$$

It can be verified that this formula realizes a layout satisfying the requirement. Similarly, we can write a formula to compute b from s and g.

6 Update and Recovery

When a client updates a data block, we must update the corresponding checksum blocks. Here, we face two challenges that remote mirroring does not. First, unlike a mirror update, the incremental calculation of a checksum is not idempotent and so must be applied exactly once. Second, a checksum protects unrelated data blocks from different sites; therefore, the update and recovery processes of a data block may interfere with those of other blocks in the same redundancy group; for example, inconsistency between a data block and its checksum affects all data blocks in the group, while inconsistency between a data block and its mirror affects no other blocks.

Therefore, we want our protocol to ensure the idempotent property of each checksum update, and to isolate as much as possible the update and recovery processes of each data block from those of others. And, as in remote mirroring cases, we also attempt to keep remote updates from degrading local write performance.

6.1 Update

6.1.1 Invariants

An important goal of our update protocol is to ensure that redundancy groups are always "consistent" and hence can be used for recovery whenever needed. Let n be the number of data blocks in a redundancy group, m be the number of checksum blocks, $d_i, 1 \leq i \leq n$ be the content of the ith data block, $c_j, 1 \leq j \leq m$ be the content of the jth checksum block, and $C_j, 1 \leq j \leq m$ be the jth checksum operation. The *group* $\{d_1, ..., d_n, c_1, ..., c_m\}$ is *consistent* if and only if $\forall j, 1 \leq j \leq m, c_j == C_j(\{d_i | 1 \leq i \leq n\})$. The checksums, $\{c_j | 1 \leq j \leq m\}$, are *consistent* with each other if and only if they belong in the same consistent group.

We maintain consistency by writing the new content of a data block (called a new *version*) in a new physical location instead of overwriting the old content in place, a technique known as *shadow paging* or *versioning*. Each new version is identified by a monotonically increasing version number. Accordingly, a checksum can

be uniquely identified by a *version vector* consisting of version numbers of the (data block) versions from which this checksum is computed.

We say that a checksum block c_j is *consistent* with a data version d_{i,v_i} and vice verse if and only if there exists a set O of versions of other data blocks in the group, i.e. $O = \{d_{k,v_k} | 1 \leq k \leq n, k \neq i\}$, such that $c_j == C_j(\{d' | d' == d_{i,v_i} || d' \in O\}$. We say that a data version d^*_{i,v_i} is *stable* at a given time if and only if all checksum sites are capable of providing a consistent checksum for that version at that time. In contrast, a data version that has not been stable is called *outstanding*.

However, the fact that every checksum site is capable of providing a consistent checksum for every data block in a redundancy group does not guarantee the group consistency, because a checksum site may not be capable of providing a checksum that is consistent with *all* data blocks. Therefore, we introduce the concept of *set consistency*. Let S be a *set* of data versions, i.e. $S = \{d_{i_l} | 1 \leq l \leq n'\}$, where n' is the size of S, $1 \leq n' \leq n$, and $\forall l, d_{i_l}$ is from data site i_l. A checksum block c_j is *consistent* with S or vice verse if and only if there exists a set O of versions of other data blocks in the group, i.e. $O = \{d_k | 1 \leq k \leq n \&\& \forall l, k \neq i_l\}$, such that $c_j == C_j(\{d' | d' \in S || d' \in O\})$.

We maintain the following two invariants in our update protocol:

1. At any time, at least one stable version of each data block exists.
2. If it is capable of providing a consistent checksum for each *individual* data version in the set $S = \{d_{i_l} | 1 \leq l \leq n'\}$, then a checksum site is capable of providing a consistent checksum for the entire S.

We can infer the following:

1. At any time, there exists a set S^* of stable data versions, i.e. $S^* = \{d^*_i | 1 \leq i \leq n\}$, and each checksum site is capable of providing a consistent checksum for each individual data version d^*_i in S^*. (Invariant 1)
2. Each checksum site j is capable of providing a consistent checksum c^*_j for the entire S^*. (Invariant 2)
3. The redundancy group $\{d^*_1, ...d^*_n, c^*_1, ...c^*_m\}$ is consistent. (Definition of group consistency)

Therefore, the redundancy group is always consistent and can be used for recovery.

We believe that the versioning approach permits a simpler, less error-prone protocol than an update-in-place approach. Because recovery never relies on blocks in transition states, we need not deal with detecting and correcting such states.

6.1.2 Two-Phase Commit

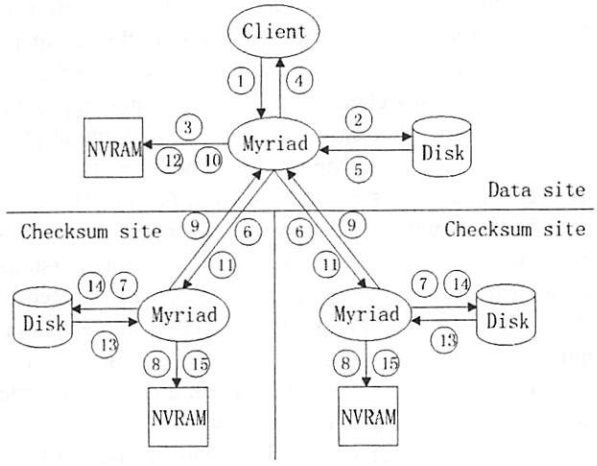
A naive way of guaranteeing at least one stable version per block (invariant 1) is to keep all old versions and their checksums. To save space, however, we should delete old versions and reclaim their physical storage as soon as a new stable version is created.

We maintain invariant 1 without storing unnecessary old versions by implementing the transition from an old stable version to a new one with a two-phase commit protocol across the data site and all checksum sites. In the *prepare phase*, each site writes enough information to local persistent storage to ensure that, in the face of system crashes and reboots, it will be capable of providing either the new data version (if it is the data site) or a consistent checksum for the new data version (if it is a checksum site). When all sites have reached the *commit point*, i.e. have completed the writes, they proceed to the *commit phase*, i.e. delete the old versions. Site/network outages may delay the communications across sites, but we ensure that the operation will proceed and the unnecessary blocks will be reclaimed once the the communications are reestablished (Section 6.1.3). The update process for a new data version will be *aborted* only if there is data loss in the redundancy group during the prepare phase, and there are not enough surviving sites to recover the new version. If the process is aborted, the new version will be deleted and the old kept. In fact, the decision for an abort cannot be made until the lost sites have been recovered (Section 6.2).

Figure 6 shows the steps in the update protocol.

When a client writes to a (logical) data block, we create a new outstanding data version by writing the data into a free physical block and logging the outstanding version with the new physical address. Here, we take advantage of the mapping from logical to physical blocks maintained by the local storage system (Section 5). Though subsequent client operations will be performed on the newest version only, the old versions are kept because they may still be needed for recovering data on other sites.

Next, the delta between the newest and the second newest data versions is sent to all checksum sites in an update request. (Consecutive writes to the same block can be collapsed into one update request unless they straddle a "sync" operation. See Section 6.3.) Each checksum site writes the data delta into a free block in its local disk, logs the outstanding version with the address of the delta, and replies to the data site that it is now capable of providing a checksum for the new data version. In addition, since the delta of each data block in the same group is stored independently, the checksum site is capable of computing a new checksum with the old checksum and any combination of the deltas; therefore,



Messages and operations:

1. MyriadWrite(laddr, new_data)
2. DiskWrite(new_paddr, new_data)
3. AddToLog(laddr, new_vernum, new_paddr)
4. WriteCompleted
5. old_data ← DiskRead(old_paddr), delta ← old_data ⊕ new_data
6. UpdateRequest(data_site_id, laddr, new_vernum, delta)
7. DiskWrite(delta_addr, delta)
8. AddToLog(group_id, data_site_id, new_vernum, delta_addr)
9. UpdateReply(checksum_site_id, laddr, new_vernum)
10. UpdateMap(laddr, new_paddr, new_vernum), FreeBlock(old_paddr)
11. CommitRequest(data_site_id, laddr, new_vernum)
12. RemoveFromLog(laddr, new_vernum)
13. old_checksum ← DiskRead(checksum_addr), new_checksum ← ChecksumOp(old_checksum, delta)
14. DiskWrite(checksum_addr, new_checksum)
15. FreeBlock(delta_addr), RemoveFromLog(group_id, data_site_id, new_vernum)

Meanings of variable names:

- laddr: logical address of a data block
- paddr: physical address of a data block
- delta_addr: address of the newly allocated physical block for the delta
- checksum_addr: address of the checksum block

Figure 6: The update protocol at a glance. The disks in the diagram are used to store data only. The NVRAM is used to store metadata and backed by redundant disks, which are not shown in the diagram.

invariant 2 is maintained.

Once it receives update replies from all checksum sites, the data site makes the new version the stable version by pointing the logical-to-physical map entry to the physical address of the new version, frees the physical block that holds the old stable version, sends a commit request to each checksum site, and then removes the outstanding version from the log. When it receives the commit request, a checksum site computes the new checksum with the new data delta, writes it on disk, deletes the delta, and removes the outstanding version from the log.

6.1.3 Redo Logs

In order for an update to proceed after temporary site/network outages, we need to maintain for each logical disk on data sites a redo log, indexed by logical addresses. Each entry in the log contains a list of data structures for outstanding versions of the block. Each entry in the list contains the outstanding version number, the physical address, and the status of each checksum site regarding the remote update of this version. The status is "ready" if an update reply from the checksum site has been received, or "pending" otherwise.

We also need to maintain for each logical checksum disk on checksum sites a redo log, indexed by redundancy group ids. Each entry in the log contains a list of data structures for outstanding data versions in the group. Each entry in the list contains the data site and outstanding version number, and the address where the data delta is stored.

The redo logs, together with other metadata such as the logical-to-physical maps, need to be stored in a permanent storage system that provides higher reliability than those for regular data, since we would like to avoid the cases where the loss of metadata causes surviving data to be inaccessible. The metadata also needs to be cached in memory for fast reads and batched writes to disk. In an ideal configuration, the metadata ought to be cached in non-volatile memory and backed by triple mirroring disks, assuming that regular data is stored on RAID-5 disks.

During each operation, e.g. client read, client write or recovery read, the redo log is always looked up before the logical-to-physical map, so that the newest version is used in the operation.

The redo log will be scanned during a system reboot or a network reconnect. An entry in the log on a data site is created after an outstanding data version is written to disk, and deleted after update replies are received from all checksum sites. Therefore, the presence of such an entry during a system reboot or network reconnect indicates that an update request with the data delta should be resent to all checksum sites with a "pending" status.

On a checksum site, an entry in the redo log is created after the data delta is written to disk, and deleted after the checksum is recomputed with the delta and stored on disk. Therefore, the presence of such an entry during a system reboot or network reconnect indicates that an update reply should be resent to the data site.

The redo logs can also be used to detect duplicate messages and hence to ensure idempotent updates. Upon receiving an update request with an outstanding version number, a checksum site first checks if the version number already exists in its redo log. If it does, the checksum site learns that it has already committed the data delta, therefore resends an update reply to the data site. Upon receiving an update reply, the data site first looks up the redo log for a corresponding entry. If none is found, the data sites learns that the outstanding version has already been committed locally, therefore resends a commit request to the checksum site. Upon receiving a commit request, a checksum tries to locate a corresponding entry in the outstanding log. If it fails to do, it learns that the version has already been committed, and therefore ignores the request.

6.2 Recovery

Cross-site recovery is initiated when a site loses data that cannot be recovered using local redundancy. The recovered data can be stored either on the same site as the lost data, or on a new site if the old site is destroyed completely. In either case, the site where the recovered data is to be stored serves as the *coordinator* during the recovery process.

We assume that metadata (e.g., the redo logs and logical-to-physical maps) on both data and checksum sites is stored with high local reliability, such that it will not be lost unless a site suffers a complete disaster. We do not attempt to recover metadata from remote sites. In the event of a site disaster, we rebuild metadata from scratch.

In the beginning of a recovery process, the coordinator determines the logical addresses of the data blocks to recover. If a site loses some storage devices but not the metadata, it can determine the addresses of blocks on the lost devices by scanning its logical-to-physical map. If a site is completely destroyed, all blocks in the address range from 0 to the capacity of the lost logical disk need to be recovered.

To reconstruct a lost data block d_i, the coordinator first determines which checksums (identified by a version vector) from surviving checksum sites and which data versions from surviving data sites to use. Then, the coordinator requests those versions and compute the lost data.

The version vector is determined in the following way. The coordinator requests the newest version numbers of the lost block d_i from surviving checksum sites, and the stable version numbers of other blocks in the same group from surviving data sites. (Such requests are referred to *VersionRequest* below.) If k data blocks in the group are lost, the *newest recoverable* version of block d_i is the one for which at least k checksum sites are capable of providing a consistent checksum. It is guaranteed that at least one such version, i.e. the stable version, exists as long as k checksum sites survive (Section 6.1.1). (If fewer than k checksum sites survive, recovery is simply impossible under the given encoding.) The stable versions of other data blocks in the group are also guaranteed to exist. The coordinator requests the explicit stable version numbers from data sites because the checksum sites may transiently have an old stable version and consider the new stable version to be outstanding still.

The version vector of the checksums consists of the newest recoverable version of the lost block d_i and the stable versions of other data blocks. Upon replying to a *VersionRequest*, a surviving site temporarily suspends the commit operations for the block involved. This way, the version selected by the coordinator will still be available by the time it is requested in the second step. Client writes and remote updates of the involved block are not suspended; only the deletion of the old stable version is postponed.

Once the version vector is determined, the coordinator requests the selected data versions and checksums from the surviving sites, reconstructs the lost data block from the returned blocks, and writes it onto a local disk. After sending the requested block, each surviving site can resume the commit operations for that block. If the data reconstruction did not complete for any reason, e.g. the coordinator crashes, a re-selection of version vector is necessary.

Finally, the coordinator attempts to synchronize all checksum sites with the recovered data version, i.e. to commit the recovered version and to delete other (older or newer) versions if there are any. The coordinator uses the redo log to ensure eventual synchronization in the face of site/network outages.

The protocol described above is for the recovery of a data block. The recovery of a checksum block is similar, with small variations in the determination of group numbers, in the selection of data versions and in the final synchronization. We omit the details here in the interest of space.

6.3 Serialization of Remote Updates

We need to ensure that consecutive writes to the same data block are committed on both data and checksum sites in the same order as the write operations return to clients. This can be done by sending the update and com-

mit requests for the same block in the ascending order of their version numbers.

Applications sometimes need explicit serialization of writes as well. For example, a file system may want to ensure that a data block is written before its inode is updated to point to the block. In the presence of buffer caches in storage systems, the serialization needs to be done via a "sync" bit in a block write request or a separate "sync" command (e.g. the SYNCHRONIZE CACHE operation in SCSI-2 [13]); both sync requests cause specified blocks to be flushed from cache to disk before the requests are completed. It is required that writes issued after a sync request are perceived to take effect after the sync request does, in the face of system crashes.

Unfortunately, it may not be practical to require that remote checksums be committed as well before a sync request is completed. The long latency in WAN communication may be unacceptable to certain applications. If a checksum site is unreachable, the sync could be delayed indefinitely.

Therefore, we relax the semantics for sync requests in the cross-site redundancy context for better performance and availability. In our system, a sync request is completed after the requested data has reached local storage, but before its delta reaches the checksum sites. In order to prevent inconsistency upon recovery caused by out-of-order writes, we guarantee that writes following a sync request are propagated to the checksum sites only after the data in the sync request has been committed on the checksum sites. Therefore, we can collapse the update requests for consecutive writes to the same data block and propagate them as one request only if those writes are between two consecutive sync operations.

The serialization during a redo process after a system crash or network outage can be enforced by resending update requests in the ascending order of version numbers. This indicates that version numbers of all data blocks on the same logical disk need to be serializable.

We do not attempt to guarantee cross-site serialization because Myriad is designed for independent applications per site (Section 2).

6.4 Performance Implications

A client write operation involves a single disk write of the new data and a few updates to the metadata in non-volatile memory; therefore, we do not expect the client to observe significant increase in write latency. A complete remote update requires the following additional operations on the data site: a disk read, an xor operation and several updates to the metadata in non-volatile memory. It also requires the transmission of a block over the WAN, and the following operations on each checksum site: a disk read, a checksum computation, two disk writes, and several updates to the metadata in non-volatile memory. Therefore, for a Myriad system with n data sites, the write bandwidth on each data site is limited by the minimum of the following: data site disk bandwidth divided by 2, WAN bandwidth, and checksum site disk bandwidth divided by 3. We expect the WAN bandwidth to be the limiting factor.

The consumption of WAN bandwidth in our scheme is comparable to that of a mirroring scheme able to survive the same number of site losses. As Figure 6 shows, if there are k checksum sites, for each logical block written, an update request with the delta (of the size of a block) is sent k times, once to each checksum site. A system with k remote mirrors would also require sending a newly written block of data k times, once to each mirror site. We also send k commit requests and expect a mirroring scheme to do the same if it also uses two-phase commit to guarantee cross-mirror consistency. Similar optimizations (e.g., collapsing consecutive writes to the same block) can apply in both cases.

6.5 Storage Overhead

As discussed earlier in this section, we need a logical-to-physical map for each logical disk on data sites. There is an entry in the map for each logical data block, and the map is indexed by logical block address. Each entry in the map contains the physical address of the stable version, and the stable version number. We also need a logical-to-physical map for each logical disk on checksum sites. There is an entry in the map for each redundancy group, and the map is indexed by group id. Each entry in the map contains the physical address of the checksum block, and the stable version numbers of data blocks in the group.

Assume that each redundancy group consists of n data and m checksum blocks. For every n data blocks, the storage required for the map entries is $n \times (sizeof(paddr) + sizeof(vernum))$ bytes on the n data sites, and $m \times (sizeof(paddr) + n \times sizeof(vernum))$ bytes on the m checksum sites. Therefore, the overall storage overhead for the maps is $(1 + \frac{m}{n}) \times sizeof(paddr) + (1 + m) \times sizeof(vernum)$ bytes per data block. For example, with $n = 3$, $m = 2$, $sizeof(paddr) = 4$ bytes, and $sizeof(vernum) = 4$ bytes, it would amount to 18.7 bytes per data block. The storage overhead for the maps is roughly 0.028% for a block size of 64 KB and 0.45% for a block size of 4 KB.

We expect the maps to be much larger than the redo logs because the latter contains only blocks "in transition". The number of such blocks depends on the burstiness of client writes and on the difference between the local disk bandwidth and the WAN bandwidth.

7 Related Work

Myriad is most related to the distributed RAID algorithm proposed by Stonebraker and Schloss [19]. Like Myriad, they envision independent local storage systems on geographically separate sites protecting one another's data with a redundancy scheme other than mirroring. However, a key difference is that their redundancy groups consist of physical blocks while Myriad's consist of logical blocks. Physical blocks are overwritten in place by client and redundancy update operations. Thus, redundancy groups could become inconsistent, though their recovery procedure can detect this and retry. Also, local write latency is roughly doubled because a local write cannot return until after the old data are read (to compute the delta) *and* subsequently overwritten with the new data. Myriad avoids these by forming redundancy groups with logical blocks that may have multiple versions coexisting simultaneously. During a redundancy update, the old versions of data and checksum are not affected and remain consistent. And a local write can return immediately after a single I/O (to write the new data version); the old data can be read and the delta computed later.

Striped distributed file systems such as Swift [15], Zebra [11] and xFS [20, 3] are related to Myriad in that they also keep data and parity blocks on multiple storage servers. However, they have different a technology assumption, goal, and hence data layout. They are designed for servers connected by a high-performance LAN, while in our case servers are connected by a (relatively) low-performance WAN. Since the LAN has low latency and high bandwidth, these systems stripe the data blocks in a file across servers to maximize read/write bandwidth via server parallelism. For Myriad, since moving data over a WAN is slow, related data reside on the same site and client accesses are always local. Checksum blocks are computed from unrelated data blocks on different sites only for disaster tolerance.

TickerTAIP is a disk array architecture that distributes controller functions across loosely coupled processing nodes [4]. It is related to Myriad in that multiple nodes cooperate to perform a client write and the corresponding parity update. TickerTAIP uses a two-phase commit protocol to ensure write atomicity, and proceeds with a write when enough data has been replicated in more than one node's *memory*. After a node crashes and reboots, the replicated data can be copied to that node so that the operation will complete eventually. Myriad commits the write when sufficient data has been written to *permanent storage* so that each site can join a consistent group upon crash and reboot without requesting data from other sites first. The difference results from our attempt to avoid as much as possible WAN communications. In another respect, TickerTAIP preserves partial ordering of reads and writes by offering an interface for each request to *explicitly* list other requests that it depends on. It then manages request sequencing by modeling each request as a state machine. Myriad also attempts to preserve partial ordering on the same site, but only using standard interfaces, e.g. the SYNCHRONIZE CACHE operation in SCSI-2 [13]. All reads and writes following a sync operation *implicitly* depend on that operation. No cross-site sequencing is supported because Myriad is designed for independent applications per site. As a result, the management of sequencing in Myriad is much simpler than that in TickerTAIP.

Aspects of Myriad's design use classic techniques widely used elsewhere. In Myriad, clients access blocks using logical addresses, while the storage system decides which physical block(s) actually contain the data. We exploit this logical-physical separation to keep physical blocks of data and checksums consistent during redundancy updates. It has been used for many other purposes: in Loge to improve disk write performance by allowing new data to be written to any convenient location on a disk surface [9], in Mime to enable a multi-disk storage subsystem to provide transaction-like capabilities to its clients [5], in Logical Disk to separate file management from disk management and thus improve the structure and performance of file systems [6], and in the HP AutoRAID hierarchical storage system to allow migration of data between different RAID levels (namely RAID-1 and RAID-5) in a way transparent to clients [21].

Like AFRAID [17], Myriad trades a small window of data vulnerability for write performance. Specifically, Myriad updates remote redundancy information after the client write has returned. We make this design choice because client writes would otherwise be too slow. In exchange, the newly written data will be vulnerable to a site disaster before the update is completed. However, other (previously written) data blocks in the same group are not affected because we do not update in place and so the old data and checksum versions remain consistent. In contrast, the AFRAID disk array design physically overwrites data blocks without parity updates, leaving the group inconsistent until the parity block is recomputed later. Before that, *all* data blocks in the group are vulnerable to a single-disk failure. We prefer not to take a similar risk in Myriad. If we did, each site might always have some blocks that are not protected from disaster simply because other blocks (on other, separately operated sites) in the same redundancy group have just been written by their local clients and the corresponding redundancy updates are in progress. Moreover, to recompute the checksum block(s) as AFRAID does, Myriad would have to send all the data blocks in the group to the checksum site(s) *and* to carefully orchestrate the recom-

putation as an atomic operation. These are much more expensive and complex in a WAN-connected distributed system like Myriad than in a disk array with a centralized controller like AFRAID.

8 Conclusions and Future Work

The usual approach to ensuring that data in a geoplex survives site failures is to mirror the data. We have presented early results of our study in using erasure codes across sites as an alternative. Our motivation is to reduce the cost of providing data disaster tolerance while retaining much of the reliability. Our results thus far indicate that the TCO for the storage system can be reduced by 20–25% (relative to mirroring) while providing reliability far beyond a non-disaster-tolerant system. We also present a protocol for updates and recovery in a redundancy scheme based on erasure codes. Our scheme makes sense under the system and application assumptions in Section 2, although these are less general than for mirroring.

While the related idea of software RAID has been studied in various contexts, we believe that our approach is novel. In essence, we combine unrelated data blocks at different sites into redundancy groups that protect the data of their members. We assume that the data sites provide a logical disk interface to their client applications. This simplifies our protocol design by using multiple data versions instead of overwriting data blocks in place, making recovery much less error-prone.

There are several interesting future directions. While static block grouping is simple, a more dynamic scheme may offer more flexibility for site configurations. It involves maintaining maps of group ids to the logical data blocks each group consists of. Also, we would like to have a systematic proof for the correctness of the update and recovery protocol. Finally, since we use shadow paging, for now we can only exploit sequential disk access within a block. So we want a relatively large block size, which in turn makes the solution less general. It may be worthwhile to explore alternatives that rely on intelligent physical placement of data blocks.

9 Acknowledgments

Raymie Stata, Mike Burrows, and Mark Manasse contributed to our early discussion.

References

[1] N. Allen. Don't waste your storage dollars: What you need to know. Research Note COM-13-1217, Gartner Group, Stamford, CT, March 20, 2001. Available from Gartner at http://www.gartner.com/.

[2] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 62–72, Denver, CO, June 1997. IEEE Computer Society Press.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[4] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems*, 12(3):236–269, August 1994.

[5] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9 rev. 1, Hewlett-Packard Laboratories, Palo Alto, CA, November 1992.

[6] W. de Jonge, M. F. Kasshoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 15–28, Asheville, NC, December 1993.

[7] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and packs: RACS and RAPS. Technical Report MS-TR-99-85, Microsoft Research, Redmond, CA, December 1999.

[8] M. L. Dolinov, B. Hannigan, and T. Dolan. Metro's ethernet future. Forrester Research, Cambridge, MA, October 2000. Available from Forrester with registration at http://www.forrester.com/.

[9] R. M. English and A. A. Stepanov. Loge: A self-organizing disk controller. In *Proceedings of USENIX 1992 Winter Technical Conference*, pages 237–251, San Francisco, CA, January 1992.

[10] G. A. Gibson and D. A. Patterson. Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17(1–2):4–27, January–February 1993.

[11] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, June 1995.

[12] T. Kraemer, J. Berlino, J. Griffin, D. Haynes, T. Herbig, P. Stern, and A. Torres. The storage report — customer perspectives and industry evolution. Joint industry study by McKinsey & Company and Merrill Lynch, New York, NY, June 19, 2001.

[13] L. Lamers, editor. *Small Computer System Interface - 2, Revision 10L*. Computer and Business Equipment Manufacturers Association, September, 1993.

[14] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.

[15] D. D. E. Long, B. R. Montague, and L.-F. Cabrera. Swift/RAID: a distributed RAID system. Technical Report UCSC-CRL-94-06, University of California, Santa Cruz, 1994.

[16] W. W. Peterson and E. J. Weldon. *Error-correcting Codes*. The MIT Press, 2nd edition, 1972.

[17] S. Savage and J. Wilkes. AFRAID – a frequently redundant array of independent disks. In *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, San Diego, CA, January 1996.

[18] G. Schreck. Slaying the storage beast. Forrester Research, Cambridge, MA, March 2001. Available from Forrester with registration at http://www.forrester.com/.

[19] M. Stonebraker and G. A. Schloss. Distributed RAID – a new multiple copy algorithm. In *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pages 430–437, February 1990.

[20] R. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 71–78, Napa, CA, 1993.

[21] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

SnapMirror®: File System Based Asynchronous Mirroring for Disaster Recovery

Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, Shane Owara

Network Appliance Inc.
Sunnyvale, CA

{hugo, stephen, mikef, hitz, srk, owara}@netapp.com

Abstract

Computerized data has become critical to the survival of an enterprise. Companies must have a strategy for recovering their data should a disaster such as a fire destroy the primary data center. Current mechanisms offer data managers a stark choice: rely on affordable tape but risk the loss of a full day of data and face many hours or even days to recover, or have the benefits of a fully synchronized on-line remote mirror, but pay steep costs in both write latency and network bandwidth to maintain the mirror. In this paper, we argue that asynchronous mirroring, in which batches of updates are periodically sent to the remote mirror, can let data managers find a balance between these extremes. First, by eliminating the write latency issue, asynchrony greatly reduces the performance cost of a remote mirror. Second, by storing up batches of writes, asynchronous mirroring can avoid sending deleted or overwritten data and thereby reduce network bandwidth requirements. Data managers can tune the update frequency to trade network bandwidth against the potential loss of more data. We present SnapMirror, an asynchronous mirroring technology that leverages file system snapshots to ensure the consistency of the remote mirror and optimize data transfer. We use traces of production filers to show that even updating an asynchronous mirror every 15 minutes can reduce data transferred by 30% to 80%. We find that exploiting file system knowledge of deletions is critical to achieving any reduction for no-overwrite file systems such as WAFL and LFS. Experiments on a running system show that using file system metadata can reduce the time to identify changed blocks from minutes to seconds compared to purely logical approaches. Finally, we show that using SnapMirror to update every 30 minutes increases the response time of a heavily loaded system only 22%.

1 Introduction

As reliance on computerized data storage has grown, so too has the cost of data unavailability. A few hours downtime can cost from thousands to millions of dollars depending on the size of the enterprise and the role of the data. With increasing frequency, companies are instituting disaster recovery plans to ensure appropriate data availability in the event of a catastrophic failure or disaster that destroys a site (e.g. flood, fire, or earthquake). It is relatively easy to provide redundant server and storage hardware to protect against the loss of physical resources. Without the data, however, the redundant hardware is of little use.

The problem is that current strategies for data protection and recovery offer either inadequate protection, or are too expensive in performance and/or network bandwidth. Tape backup and restore is the traditional approach. Although favored for its low cost, restoring from a nightly backup is too slow and the restored data is up to a day old. Remote synchronous and semi-synchronous mirroring are more recent alternatives. Mirrors keep backup data on-line and fully synchronized with the primary store, but they do so at a high cost in performance (write latency) and network bandwidth. Semi-synchronous mirrors can reduce the write-latency penalty, but can result in inconsistent, unusable data unless write ordering across the entire data set, not just within one storage device, is guaranteed. Data managers are forced to choose between two extremes: synchronized with great expense or affordable with a day of data loss.

In this paper, we show that by letting a mirror volume lag behind the primary volume it is possible to reduce substantially the performance and network costs of maintaining a mirror while bounding the amount of data loss. The greater the lag, the greater the data loss, but the cheaper the cost of maintaining the mirror. Such asynchronous mirrors let data managers tune their systems to strike the right balance between potential data loss and cost.

We present SnapMirror, a technology which implements asynchronous mirrors on Network Appliance filers. SnapMirror periodically transfers self-consistent snapshots of the data from a source volume to the destination volume. The mirror is on-line, so disaster recov-

SnapMirror, NetApp, and WAFL are registered trademarks of Network Appliance, Inc.

ery can be instantaneous. Users set the update frequency. If the update frequency is high, the mirror will be nearly current with the source and very little data will be lost when disaster strikes. But, by lowering the update frequency, data managers can reduce the performance and network cost of maintaining the mirror at the risk of increased data loss.

There are three main problems in maintaining an asynchronous mirror. First, for each periodic transfer, the system must determine which blocks need to be transferred to the mirror. To obtain the bandwidth reduction benefits of asynchrony, the system must avoid transferring data which is overwritten or deleted. Second, if the source volume fails at any time, the destination must be ready to come on line. In particular, a half-completed transfer can't leave the destination in an unusable state. Effectively, this means that the destination must be in, or at least recoverable to, a self-consistent, state at all times. Finally, for performance, disk reads on the source and writes on the destination must be efficient.

In this paper, we show how SnapMirror leverages the internal data structures of NetApp's WAFL® file system [Hitz94] to solve these problems. SnapMirror leverages the active block maps in WAFL's snapshots to quickly identify changed blocks and avoid transferring deleted blocks. Because SnapMirror transfers self-consistent snapshots of the file system, the remote mirror is always guaranteed to be in a consistent state. New updates appear atomically. Finally, because it operates at the block level, SnapMirror is able to optimize its data reads and writes.

We show that SnapMirror's periodic updates transfer much less data than synchronous block-level mirrors. Update intervals as short as 1 minute are sufficient to reduce data transfers by 30% to 80%. The longer the period between updates, the less data needs to be transferred. SnapMirror allows data managers to optimize the tradeoff of data currency against cost for each volume.

In this paper, we explore the interaction between asynchronous mirroring and no-overwrite file systems such as LFS [Rosenblum92] and WAFL. We find that asynchronous block-level mirroring of these file systems does not transfer less data synchronous mirroring. Because these file systems do not update in place, logical overwrites become writes to new storage blocks. To gain the data reduction benefits of asynchrony for these file systems, it is necessary to have knowledge of which blocks are active and which have been deallocated and are no longer needed. This is an important observation since many commercial mirroring products are implemented at the block level.

1.1 Outline for remainder of paper

We start, in Section 1.2, with a discussion of the requirements for disaster recovery. We go on in Sections 1.3 and 1.4 to discuss the shortcomings of tape-based recovery and synchronous remote mirroring. In Section 2, we review related work. We present the design and implementation of SnapMirror in Section 3. In Section 4, we use system traces to study the data reduction benefits of asynchronous mirroring with file system knowledge. Then, in Section 5, we compare SnapMirror to asynchronous mirroring at the logical file level. Section 6, presents experiments measuring the performance of our SnapMirror implementation running on a loaded system. Conclusion, acknowledgments, and references are in Sections 7, 8, and 9.

1.2 Requirements for Disaster Recovery

Disaster recovery is the process of restoring access to a data set after the original was destroyed or became unavailable. Disasters should be rare, but data unavailability must be minimized. Large enterprises are asking for disaster recovery techniques that meet the following requirements:

Recover quickly. The data should be accessible within a few minutes after a failure.

Recover consistently. The data must be in a consistent state so that the application does not fail during the recovery attempt because of a corrupt data set.

Minimal impact on normal operations. The performance impact of a disaster recovery technique should be minimal during normal operations.

Up to date. If a disaster occurs, the recovered data should reflect the state of the original system as closely as possible. Loss of a day or more worth of updates is not acceptable in many applications.

Unlimited distance. The physical separation between the original and recovered data should not be limited. Companies may have widely separated sites and the scope of disasters such as earthquakes or hurricanes may require hundreds of miles of separation.

Reasonable cost. The solution should not require excessive cost, such as many high-speed, long-distance links (e.g. direct fiber optic cable). Preferably, the link should be compatible with WAN technology.

1.3 Recovering from Off-line Data

Traditional disaster recovery strategies involve loading a saved copy of the data from tape onto a new server in a different location. After a disaster, the most recent full backup tapes are loaded onto the new server. A series of nightly incremental backups may follow the

full backup to bring the recovered volume as up-to-date as possible. This worked well when file systems were of moderate size and when the cost of a few hours of down-time was acceptable, provided such events were rare.

Today, companies are taking advantage of the 60% compound annual growth rate in disk drive capacity [Growchowski96] and file system size is growing rapidly. Terabyte storage systems are becoming commonplace. Even with the latest image dump technologies [Hutchinson99], data can only be restored at a rate of 100-200 GB/hour. If disaster strikes a terabyte file system, it will be off line for at least 5-10 hours if tape-based recovery technologies are used. This is unacceptable in many environments.

Will technology trends solve this problem over time? Unfortunately, the trends are against us. Although disk capacities are growing 60% per year, disk transfer rates are growing at only 40% per year [Grochowski96]. It is taking more, not less, time to fill a disk drive even in the best case of a purely sequential data stream. In practice, even image restores are not purely sequential and achieved disk bandwidth is less than the sequential ideal. To achieve timely disaster recovery, data must be kept on-line and ready to go.

1.4 Remote Mirroring

Synchronous remote mirroring immediately copies all writes to the primary volume to a remote mirror volume. The original transfer is not acknowledged until the data is written to both volumes. The mirror gives the user a second identical copy of the data to fall back on if the primary file system fails. In many cases, both copies of the data are also locally protected by RAID.

The down side of synchronous remote mirroring is that it can add a lot of latency to I/O write operations. Slower I/O writes slow down the server writing the data. The extra latency results first from serialization and transmission delays in the network link to the remote mirror. Longer distances can bloat response time to unacceptable levels. Second, unless there is a dedicated high-speed line to the remote mirror, network congestion and bandwidth limitations will further reduce performance. For these reasons, most synchronous mirroring implementations limit the distance to the remote mirror to 40 kilometers or less.

Because of its performance limitations, synchronous mirroring implementations sometimes slightly relax strict synchrony, to allow a limited number of source I/O operations to proceed before waiting for acknowledgment of receipt from the remote site[1]. Although this approach can reduce I/O latency, it does not reduce the link bandwidth needed to keep up with the writes. Further, the improved performance comes at the cost of some potential data loss in the event of a disaster.

A major challenge for non-synchronous mirroring is ensuring the consistency of the remote data. If writes arrive out-of-order at the remote site, the remote copy of the data may appear corrupted to an application trying to use the data after a disaster. If this occurs, the remote mirroring will have been useless since a full restore from tape will probably be required to bring the application back on line. The problem is especially difficult when a single data set is spread over multiple devices and the mirroring is done at the device level. Although each device guarantees in-order delivery of its the data, there may be no ordering guarantees among the devices. In a rolling disaster, one in which devices fail over a period of time (imagine fire spreading from one side of the data center to the other), the remote site may receive data from some devices but not others. Therefore, whenever synchrony is relaxed, it is important that it be coordinated at a high enough level to ensure data consistency at the remote site.

Another important issue is keeping track of the updates required on the remote mirror should it or the link between the two systems become unavailable. Once the modification log on the primary system is filled, the primary system usually abandons keeping track of individual modifications and instead keeps track of updated regions. When the destination again becomes available, the regions are transferred. Of course, the destination file system may be inconsistent while this transfer is taking place, since file system ordering rules may be violated, but it's better than starting from scratch.

2 Related Work

There are other ways to provide disaster recovery besides restore from tape and synchronous mirroring. One is server replication.

Server replication is another approach to providing high availability. Coda is one example of a replicated file system [Kistler93]. In Coda, the clients of a file server are responsible for writing to multiple servers. This approach is essentially synchronous logical-level mirroring. By putting the responsibility for replication on the clients, Coda effectively off-loads the servers. And, because clients are aware of the multiple servers, recovery from the loss of a server is essentially instantaneous. However, Coda is not designed for replication over a WAN. If the WAN connecting a client to a remote server

1. EMC's SRDF™ in semi-synchronous mode or Storage Computer's Omniforce® in log synchronous mode.

is slow or congested, the client will feel a significant performance impact. Another difference is that where Coda leverages client-side software, SnapMirror's goal is to provide disaster recovery for the file servers without client side modifications.

Earlier, we mentioned that SnapMirror leverages file system metadata to detect new data since the last update of the mirror. But, there are many other approaches.

At the logical file system level, the most common approach is to walk the directory structure checking the time that files were last updated. For example, the UNIX dump utility compares the file modify times to the time of the last dump to determines which files it should write to an incremental dump tape. Other examples of detecting new data at the logical level include programs like rdist and rsync [Tridgell96]. These programs traverse both the source and destination file systems, looking for files that have been more recently modified on the source than the destination. The rdist program will only transfer whole files. If one byte is changed in a large database file, the entire file will be transferred. The rsync program works to compute a minimal range of bytes that need be transferred by comparing checksums of byte ranges. It uses CPU resources on the source server to reduce network traffic. Compared to these programs SnapMirror does not need to traverse the entire file system or do checksums to determine the block differences between the source and destination. On the other hand, SnapMirror needs to be tightly integrated with the file system whereas approaches which operate at the logical level are more general.

Another approach to mirroring, adopted by databases such as Oracle, is to write a time-stamp in a header in each on-disk data block. The time-stamp enables Oracle to determine if a block needs to be backed up by looking only at the relatively small header. This can save a lot of time compared to approaches which must perform check sums on the contents of each block. But, it still requires each block to be scanned. In contrast, SnapMirror uses file system data structures as an index to detect updates. The total amount of data examined is similar in the two cases, but the file system structures are stored more densely and consequently the number of blocks that must be read from disk is much smaller.

3 SnapMirror Design and Implementation

SnapMirror is an asynchronous mirroring package currently available on Network Appliance file servers. Its design goal was to meet the data protection needs of large-scale systems. It provides a read-only, on-line, replica of a source file system. In the event of disaster, the replica can be made writable, replacing the original source file system.

Periodically, SnapMirror reflects changes in the source volume to the destination volume. It replicates the source at a block-level, but uses file system knowledge to limit transfers to blocks that are new or modified and that are still allocated in the file system. SnapMirror does not transfer blocks which were written but have since been overwritten or deallocated.

Each time SnapMirror updates the destination, it takes a new snapshot of the source volume. To determine which blocks need to be sent to the destination, it compares the new snapshot to the snapshot from the previous update. The destination jumps forward from one snapshot to the next when each transfer is completed. Effectively, the entire update is atomically applied to the destination volume. Because the source snapshots always contain a self-consistent, point-in-time image of the entire volume or file system, and these snapshots are applied atomically to the destination, the destination always contains a self-consistent, point-in-time image of the volume. SnapMirror solves the problem of ensuring destination data consistency even when updates are asynchronous and not all writes are transferred so ordering among individual writes cannot be maintained.

The system administrator sets SnapMirror's update frequency to balance the impact on system performance against the lag time of the mirror.

3.1 Snapshots and the Active Map File

SnapMirror's advantages lie in its knowledge of the Write Anywhere File Layout (WAFL) file system and its snapshot feature [Hitz94], which runs on top of Network Appliance's file servers. WAFL is designed to have many of the same advantages as the Log Structured File System (LFS) [Rosenblum92]. It collects file system block modification requests and then writes them to an unused group of blocks. WAFL's block allocation policy is able to fit new writes in among previously allocated blocks, and thus it avoids the need for segment-cleaning. WAFL also stores all metadata in files, like the Episode file system [Chutani92]. This allows updates to write metadata anywhere on disk, in the same manner as regular file blocks.

WAFL's on-disk data structure is a tree that points to all data and metadata. The root of the tree is called the *fsinfo block*. A complete and consistent version of the file system can be reached from the information in this block. The fsinfo block is the only exception to the no-overwrite policy. Its update protocol is essentially a database-like transaction; the rest of the file system image must be consistent whenever a new fsinfo block overwrites the old. This insures that partial writes will never corrupt the

file system.

It is easy to preserve a consistent image of a file system, called a *snapshot*, at any point in time, by simply saving a copy of the information in the fsinfo block and then making sure the blocks that comprise the file system image are not reallocated. Snapshots will share the block data that remains unmodified with the active file system; modified data are written out to unallocated blocks. A snapshot image can be accessed through a pointer to the saved fsinfo block.

WAFL maintains the block allocations for each snapshot in its own *active map file*. The active map file is an array with one allocation bit for every block in the volume. When a snapshot is taken, the current state of the active file system's active map file is frozen in the snapshot just like any other file. WAFL will not reallocate a block unless the allocation bit for the block is cleared in every snapshot's active map file. To speed block allocations, a summary active map file maintains for each block, the logical-OR of the allocation bits in all the snapshot active map files.

3.2 SnapMirror Implementation

Snapshots and the active map file provide a natural way to find out block-level differences between two instances of a file system image. SnapMirror also uses such block-level information to perform efficient block-level transfers. Because the mirror is a block-by-block replica of the source, it is easy to turn it into a primary file server for users, should disaster befall the source.

3.2.1 Initializing the Mirror

The destination triggers SnapMirror updates. The destination initiates the mirror relationship by requesting an initial transfer from the source. The source responds by taking a *base reference snapshot* and then transferring all the blocks that are allocated in that or any earlier snapshot, as specified in the snapshots' active map files. Thus, after initialization, the destination will have the same set of snapshots as the source. The base snapshot serves two purposes: first, it provides a reference point for the first update; second, it provides a static, self-consistent image which is unaffected by writes to the active file system during the transfer.

The destination system writes the blocks to the same logical location in its storage array. All the blocks in the array are logically numbered from 1 to N on both the source and the destination, so the source and destination array geometries need not be identical. However, because WAFL optimizes block layout for the underlying array geometry, SnapMirror performance is best when the source and destination geometries match and the optimizations apply equally well to both systems. When the block transfers complete, the destination writes its new fsinfo block.

3.2.2 Block-Level Differences and Update Transfers

Part of the work involved in any asynchronous mirroring technique is to find the changes that have occurred in the primary file system and make the same changes in another file system. Not surprisingly, SnapMirror uses WAFL's active map file and reference snapshots to do this as shown in Figure 1.

When a mirror has an update scheduled, it sends a message to the source. The source takes an *incremental reference snapshot* and compares the allocation bits in the active map files of the base and incremental reference snapshots. This active map file comparison follows the following rules:

If the block is not allocated in either active map, it is **unused**. The block is not transferred. It did not exist in the old file system image, and is not in use in the new one. Note that it could have been allocated and deallocated between the last update and the current one.

If the block is allocated in both active maps, it is **unchanged**. The block is not transferred. By the file system's no-overwrite policy, this block's data has not changed. It could not have been overwritten, since the old reference snapshot keeps the block from being re-allocated.

If the block is only allocated in the base active map, it has been **deleted**. The block is not transferred. The data it contained has either been deleted or changed.

If the block is only allocated in the incremental active map, it has been **added**. The block is transferred. This means that the data in this block is either new or an updated version of an old block.

Note that SnapMirror does not need to understand whether a transferred block is user data or file system metadata. All it has to know is that the block is new to the file system since the last transfer and therefore it should be transferred. In particular, block de-allocations automatically get propagated to the mirror, because the updated blocks of the active map file are transferred along with all the other blocks.

In practice, SnapMirror transfers the blocks for all existing snapshots that were created between the base and incremental reference snapshots. If a block is newly allocated in the active maps of any of these snapshots, then it is transferred. Otherwise, it is not. Thus, the destination has a copy of all of the source's snapshots.

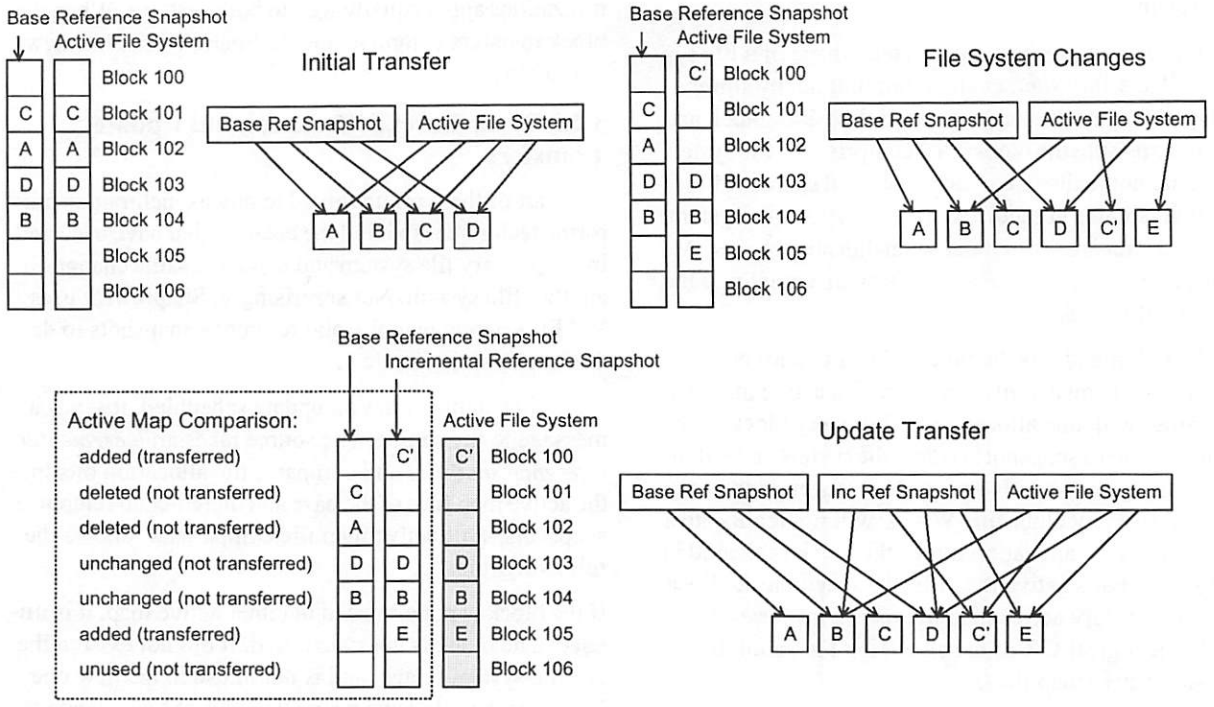


Figure 1. SnapMirror's use of snapshots to identify blocks for transfer. SnapMirror uses a base reference snapshot as point of comparison on the source and destination filers. The first such snapshot is used for the Initial Transfer. File System Changes cause the base snapshot and the active file system to diverge (C is overwritten with C', A is deleted, E is added). Snapshots and the active file system share unchanged blocks. When it is time for an Update Transfer, SnapMirror takes a new incremental reference snapshot and then compares the snapshot active maps according to the rules in the text to determine which blocks need to be transferred to the destination. After a successful update, SnapMirror deletes the old base snapshot and the incremental becomes the new base.

At the end of each transfer the fsinfo block is updated, which brings the user's view of the file system up to date with the latest transfer. The base reference snapshot is deleted from the source, and the incremental reference snapshot becomes the new base. Essentially, the file system updates are written into unused blocks on the destination and then the fsinfo block is updated to refer to this new version of the file system with is already in place.

3.2.3 Disaster Recovery and Aborted Transfers

Because a new fsinfo block (the root of the file system tree structure) is not written until all blocks are transferred, SnapMirror guarantees a consistent file system on the mirror at any time. The destination file system is accessible in a read-only state throughout the whole SnapMirror process. At any point, its active file system replicates the active map and fsinfo block of the last reference snapshot generated by the source. Should a disaster occur, the destination can be brought immediately into a writable state.

The destination can abandon any transfer in progress in response to a failure at the source end or a network partition. The mirror is left in the same state as it was before the transfer started, since the new fsinfo block is never written. Because all data is consistent with the last completed round of transfers, the mirror can be reestablished when both systems are available again by finding the most recent common SnapMirror snapshot on both systems, and using that as the base reference snapshot.

3.2.4 Update Scheduling and Transfer Rate Throttling

The destination file server controls the frequency of update through how often it requests a transfer from the source. System administrators set the frequency through a cron-like schedule. If a transfer is in progress when another scheduled time has been reached, the next transfer will start when the current transfer is complete. SnapMirror also allows the system administrator to throttle the rate at which a transfer is done. This prevents a flood of data transfers from overwhelming the disks, CPU, or network during an update.

3.3 SnapMirror Advantages and Limitations

SnapMirror meets the emerging requirements for data recovery by using asynchrony and combining file system knowledge with block-level transfers.

Because the mirror is on-line and in a consistent state at all phases of the relationship, the data is available during the mirrored relationship in a read-only capacity. Clients of the destination file system will see new updates atomically appear. If they prefer to access a stable image of the data, they can access one of the snapshots on the destination. The mirror can be brought into a writable state immediately, making disaster recovery extremely quick.

The schedule-based updates mean that SnapMirror has as much or as little impact on operations as the system administrator allows. The tunable lag also means that the administrator controls how up to date the mirror is. Under most loads, SnapMirror can reasonably transmit to the mirror many times in one hour.

SnapMirror works over a TCP/IP connection that uses standard network links. Thus, it allows for maximum flexibility in locating the source and destination filers and in the network connecting them.

The nature of SnapMirror gives it advantages over traditional mirroring approaches. With respect to synchronous mirroring, SnapMirror reduces the amount of data transferred, since blocks that have been allocated and de-allocated between updates are not transferred. And because SnapMirror uses snapshots to preserve image data, the source can service requests during a transfer. Further, updates at the source never block waiting for a transfer to the remote mirror.

The time required for a SnapMirror update is largely dependent on the amount of new data since the last update and, to some extent, on file system size. The worst-case scenario is where all data is read from and re-written to the file system between updates. In that case, SnapMirror will have to transfer all file blocks. File system size plays a part in SnapMirror performance due to the time it takes to read through the active map files (which increases as the number of total blocks increase).

Another drawback of SnapMirror is that its snapshots reduce the amount of free space in the file system. On systems with a low rate of change, this is fine, since unchanged blocks are shared between the active file system and the snapshot. Higher rates of change mean that SnapMirror reference snapshots tie up more blocks.

By design, SnapMirror only works for whole volumes as it is dependent on active map files for updates. Smaller mirror granularity could only be achieved through modifications to the file system, or through a slower, logical-level approach.

4 Data Reduction through Asynchrony

An important premise of asynchronous mirroring is that periodic updates will transfer less data than synchronous updates. Over time, many file operations become moot either because the data is overwritten or deleted. Periodic updates don't need to transfer any deleted data and only need to transfer the most recent version of an overwritten block. Essentially, periodic updates use the primary volume as a giant write cache and it has long been known that write caches can reduce I/O traffic [Ousterhout85, Baker91, Kistler93]. Still at question, though, is how much asynchrony can reduce mirror data traffic for modern file server workloads over the extended intervals of interest to asynchronous mirroring.

To answer these questions, we traced a number of file servers at Network Appliance and analyzed the traces to determine how much asynchronous mirroring would reduce data transfers as a function of update period. We also analyzed the traces to determine the importance of using the file system's active map to avoid transferring deleted blocks for WAFL as an example of no-overwrite file systems.

4.1 Tracing environment

We gathered 24 hours of traces from twelve separate file systems or volumes on four different NetApp file servers. As shown in Table 1, these file systems varied in size from 16 GB to 580 GB, and the data written over the day ranged from 1 GB to 140 GB. The blocks counted in the table are each 4 KB in size. The systems stored data from: internal web pages, engineers' home directories, kernel builds, a bug database, the source repository, core dumps, and technical publications.

In synchronous or semi-synchronous mirroring all disk writes must go to both the local and remote mirror. To determine how many blocks asynchronous mirroring would need to transfer at the end of any particular update interval, we examined the trace records and recorded in a large bit map which blocks were written (allocated) during the interval. We cleared the dirty bit whenever the block was deallocated. In an asynchronous mirroring system, this is equivalent to computing the logical-AND of the dirty map with the file system's active map and only transferring those blocks which are both dirty and still part of the active file system.

4.2 Results

Figure 2 plots the blocks that would be transferred by SnapMirror as a percentage of the blocks that would

File System Name	Filer	Description	Size (GB)	Used (GB)	Blocks Written (1000's)	Written Deleted (%)
Build1	Ecco	Source tree build space	100	68	7757	69
Cores1		Core dump storage	100	72	319	85
Bench		Benchmark scratch space and results repository	87	56	512	91
Pubs		Technical Publications	32	16	262	59
Users1		Engineering home directories	350	292	10803	78
Bug	Maglite	Bug tracking database	16	11	1465	98
Cores2		Core dump storage	550	400	11956	76
Source		Source control repository	50	36	3288	70
Cores3	Makita	Core dump storage	255	151	1582	77
Users2		Engineering home directories and corporate intranet site	580	470	13752	53
Build2	Ronco	Source tree build space	320	271	34779	80
Users3		Engineering home directories	380	323	15103	85

Table 1. Summary data for the traced file systems. We collected 24 hours of traces of block allocations (which in WAFL are the equivalent of disk writes) and de-allocations in the 12 file systems listed in the table. The 'Blocks Written' is the total number of blocks written and indicates the number of blocks that a synchronous block-level mirror would have to transfer. The 'Written Deleted' column shows the percentage of the written blocks which were overwritten or deleted. This represents the potential reduction in blocks transferred to an asynchronous mirror which is updated only once at the end of the 24-hour period. The reduction ranges from 52% to 98% and averages about 78%.

be transferred by a synchronous mirror as a function of the update period: 1 minute, 5 minutes, 15 minutes, 30 minutes, 1 hour, 6 hours, 12 hours, and 24 hours. We found that even an update interval of only 1 minute reduces the data transferred by at least 10% and by over 20% on all but one of the file systems. These results are consistent with those reported for a 30 second write-caching interval in earlier tracing studies [Ousterhout85, Baker91]. Moving to 15 minute intervals enabled asynchronous mirroring to reduce data transfers by 30% to 80% or over 50% on average. The marginal benefit of increasing the update period diminishes beyond 60 minutes. Nevertheless, extending the update period all the way to 24 hours reduces the data transferred to between 53% and 98% – over 75% on average. This represents a 50% reduction compared to an update interval of 15 minutes. Clearly, the benefits of asynchronous mirroring can be substantial.

As mentioned above, we performed the equivalent of a logical-AND of the dirty map with the file system's active map to avoid replicating deleted data. How important is this step? In conventional write-in-place file systems such as the Berkeley FFS [McKusick84], we do not expect this last step to be critical. File overwrites would repeatedly dirty the same block which would eventually only need to be transferred once. Further, because the file allocation policies of these file system often result to the reallocation of blocks recently freed, even file deletions and creations end up reusing the same set of blocks.

The situation is very different for no-overwrite file systems such as LFS and WAFL. These systems tend to avoid reusing blocks for either overwrites or new creates. Figure 3 plots the blocks transferred by SnapMirror, which takes advantage of the file system's active map to avoid transferring deallocated blocks, and an asynchronous block-level mirror, which does not, as a percentage of the blocks transferred by the synchronous mirror for a selection of the file systems. Because, most of the file systems in the study had enough free space in them to absorb all of the data writes during the day, there were essentially no block reallocations during the course of the day. For these file systems, the data reduction benefits of asynchrony would be completely lost if SnapMirror were not able to take advantage of the active maps. In the figure, the 'all other, include deallocated' line represents these results. There were two exceptions, however. Build2 wrote about 135 GB of data while the volume had only about 50 GB of free space and Source wrote about

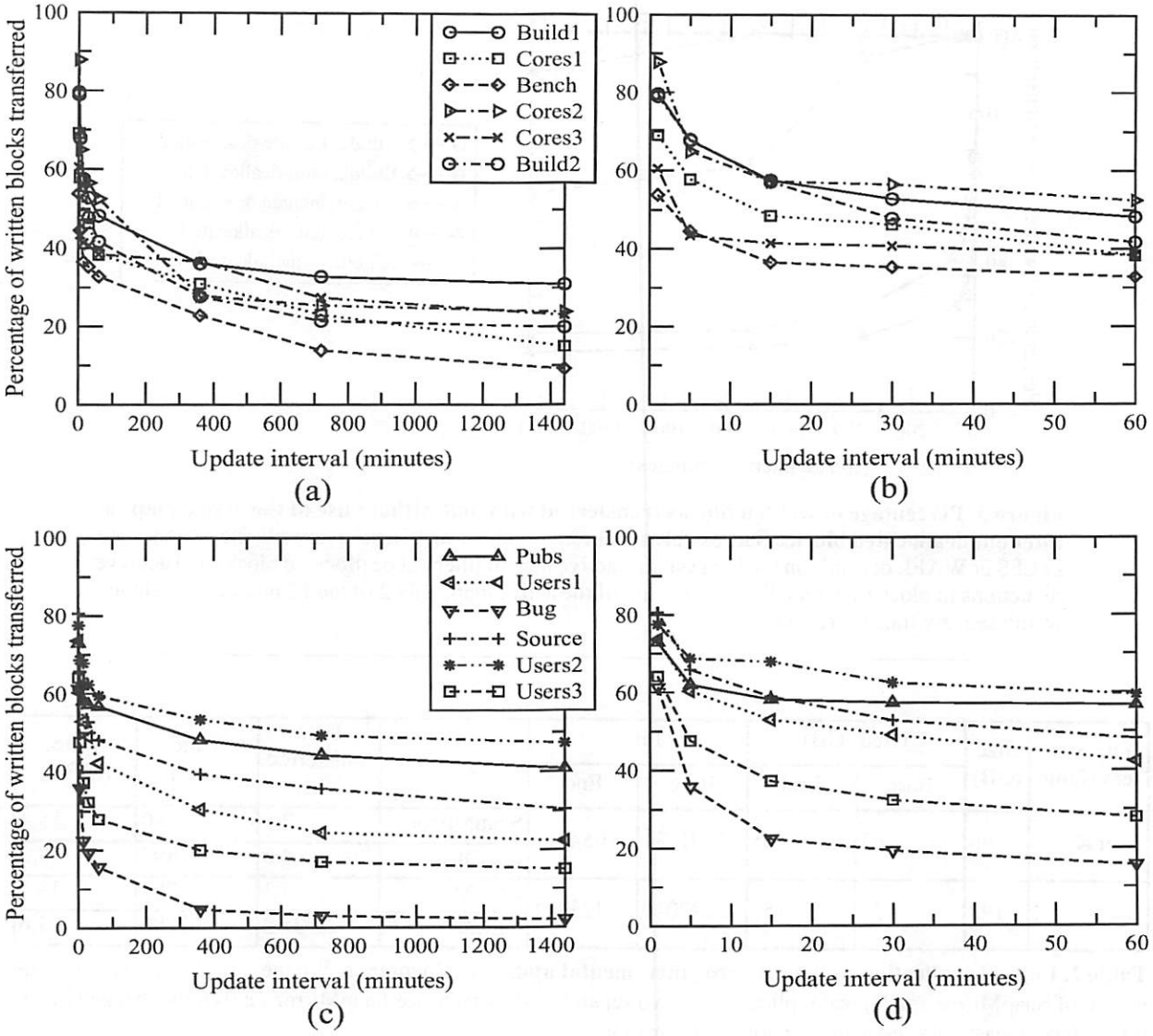


Figure 2. Percentage of written blocks transferred by SnapMirror vs. update interval. These graphs show, for each of the 12 traced systems, the percentage of written blocks that SnapMirror would transfer to the destination mirror as a function of mirror update period. Because the number of traces is large, the results are split into upper and lower pairs of graphs. The left graph in each pair (a and c) show the full range of intervals from 1 minute to 1440 minutes (24 hours). The right graphs in each pair (b and d) expand the region from 1 to 60 minutes. The graphs show that most of the reduction in data transferred occurs with an update period of as little as 15 minutes, although substantial additional reductions are possible as the interval is increased to an hour or more.

13 GB of data with only 14 GB of free space. Inevitably, in these file systems, there was some block reuse as shown in the figure. Even in these two cases, however, the use of the active map was highly beneficial. Successful asynchronous mirroring of no-overwrite file systems requires the use of the file system's active map or equivalent information.

An alternative to the block-level mirroring (with or without the active map) discussed in this section is logical or file-system level mirroring. This is the topic of the next section.

5 SnapMirror vs. Asynchronous Logical Mirroring

The UNIX dump and restore utilities can be used to implement an asynchronous logical mirror. Dump works above the operating system to identify files which need to be backed up. When performing an incremental, the utility only writes to tape the files which have been created or modified since the last incremental dump. Re-

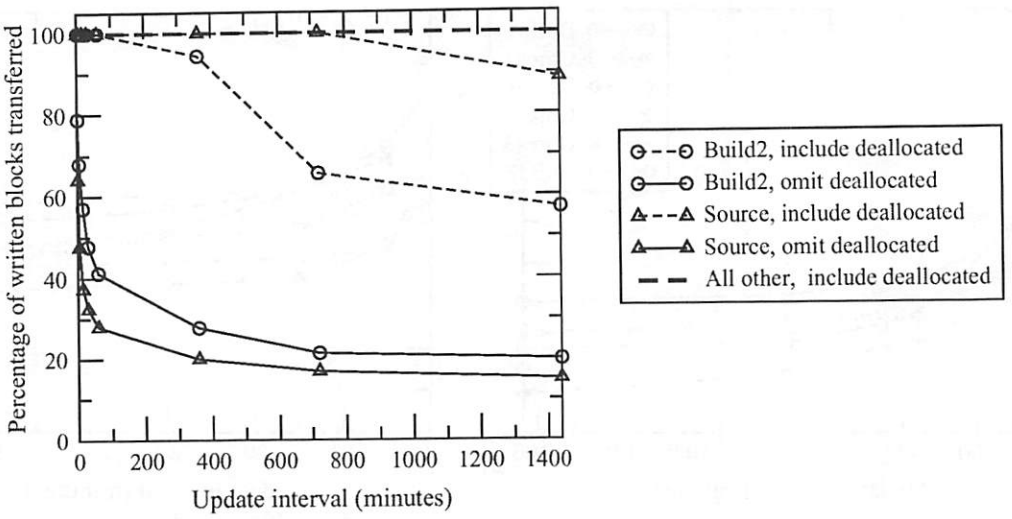


Figure 3. Percentage of written blocks transferred with and without use of the active map to filter out deallocated blocks. Successful asynchronous mirroring of a no-overwrite file system such as LFS or WAFL depends on the file system's active map to filter out deallocated blocks and achieve reductions in block transfers. Without the use of the active map, only 2 of the 12 measured systems, would see any transfer reductions.

File System Name	Size (GB)	Used (GB)		Files		System	Data transferred (GB)	Time (sec.)	Rate (MB/s)
		Base	End	Base	End				
Users4	96	63	65	1001131	1054917	SnapMirror	2.1	140	15.4
						logical	4.0	493	8.3
Users5	192	135	150	5297016	6423984	SnapMirror	15.3	797	19.7
						logical	25.2	7200	3.6

Table 2. Logical replication vs. SnapMirror incremental update performance. We measured incremental performance of SnapMirror and logical replication on two separate data sets. Since SnapMirror sends only changed blocks, it transfers at least 39% less data than logical mirroring.

store reads such incremental dumps and recreates the dumped file system. If dump's data stream is piped directly to a restore instead of a tape, the utilities effectively copy the contents of one file system to another. An asynchronous mirroring facility could periodically run an incremental dump and pipe the output to a restore running on the destination. The following set of experiments compares this approach to SnapMirror.

5.1 Experimental Setup

To implement the logical mirroring mechanism, we took advantage of the fact that Network Appliance filers include dump and restore utilities to support backup and the Network Data Management Protocol (NDMP) copy command. The command enables direct data copies from one filer to another without going through the issuing workstation. For these experiments, we configured dump to send its data over the network to a restore process on another filer. Because this code and data path are included in a shipping product, they are reasonably well tuned and the comparison to SnapMirror is fair.

To compare logical mirroring to SnapMirror, we first established and populated a mirror between two filers in the lab. We then added data to the source side of the mirror and measured the performance of the two mechanisms as they transferred the new data to the destination file system. We did this twice with two sets of data on two different sized volumes. For data, we used production full and incremental dumps of some home directory volumes. Table 2 shows the volumes and their sizes. The full dump provided the base file system. The incremental provided the new data.

We used a modified version of restore to load the incremental data into the source volume. The standard restore utility always completely overwrites files which have been updated; it never updates only the changed blocks. Had we used the standard restore, SnapMirror and the logical mirroring would both have transferred whole files. Instead, when a file on the incremental tape matched an existing file in both name and inode number, the modified restore did a block by block comparison of the new and existing files and only wrote changed blocks into the source volume. The logical mirroring mechanism, which was essentially the standard dump utility, still transferred whole files, but SnapMirror was able to take advantage of the fact that it could detect which blocks had been rewritten and thus transfer less data.

For hardware, we used two Network Appliance F760 filers directly connected via Intel GbE. Each utilized an Alpha 21164 processor running at 600 MHz, with 1024 MB of RAM plus 32 MB non-volatile write cache. For the tests run on Users4, each filer was configured with 7 FibreChannel-attached disks (18 GB, 10k rpm) on one arbitrated loop. For the tests run on Users5, each filer was configured with 14 FibreChannel-attached disks on one arbitrated loop. Each group of 7 disks was set up with 6 data disks and 1 RAID4 parity disk. All tests were run in a lab with no external load.

5.2 Results

The results for the two runs are summarized in Table 2 and Figure 4. Note that in the figure, the two sets of runs are not rendered to the same scale. The 'data scan' value for logical mirroring represents the time spent walking the directory structure to find new data. For SnapMirror, 'data scan' represents the time spent scanning the active map files. This time is essentially independent of the number of files or the amount new data but is instead a function of volume size. The number was determined by performing a null transfer on a volume of this size.

The most obvious result is that logical mirroring takes respectively 3.5 and 9.0 times longer than SnapMirror to update the remote mirror. This difference is due both to the time to scan for new data and the efficiency of the data transfers themselves. When scanning for changes, it is much more efficient to scan the active map files than to walk the directory structure. When transferring data, it is much more efficient to read and write blocks sequentially than to go through the file system code reading and writing logical blocks.

Beyond data transfer efficiency, SnapMirror is able to transfer respectively 48% and 39% fewer blocks than the logical mirror. These results show that savings from transferring only changed blocks can be substantial compared to whole file transfer.

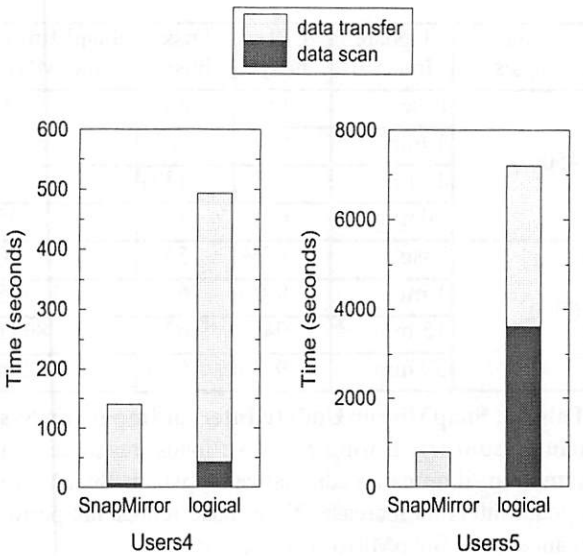


Figure 4. Logical replication vs. SnapMirror incremental update times. By avoiding directory and inode scans, SnapMirror's data scan scales much better than that of logical replication. Note: tests are not rendered on the same scale)

6 SnapMirror on a loaded system

To assess the performance impact on a loaded system of running SnapMirror, we ran some tests very much like the SPEC SFS97 [SPEC97] benchmark for NFS file servers.

In the tests, data was loaded onto the server and a number of clients submitted NFS requests at a specified aggregate rate or offered load. For these experiments, there were 48 client processes running on 6 client machines. The client machines were 167 MHz Ultra-1 Sun workstations running Solaris 2.5.1, connected to the server via switched 100bT ethernet to an ethernet NIC on the server. The server was a Network Appliance F760 filer with the same characteristics as the filers in Section 5.1. The filer had 21 disks configured in a 320 GB volume. The data was being replicated to a remote filer.

6.1 Results

After loading data onto the filer and synchronizing the mirrors, we set the SnapMirror update period to the desired value and measured the request response time over an interval of 60 minutes. Table 3 and Figure 5 report the results for an offered load of 4500 and 6000 NFS operations per second. In the table, SnapMirror data is the total data transferred to the mirror over the 60 minute

Load (ops/s)	Update Interval	CPU busy	Disk busy	SnapMirror data (MB)
4500	base	66%	34%	0
	1 min.	93%	50%	12817
	15 min.	74%	43%	6338
	30 min.	69%	40%	2505
6000	base	87%	54%	0
	1 min.	99%	67%	13965
	15 min.	94%	62%	8071
	30 min.	91%	60%	3266

Table 3. SnapMirror Update Interval Impact on System Resources. During SFS-like loads, resource consumption diminishes dramatically when SnapMirror update intervals increase. Note: base represents performance when SnapMirror is turned off.

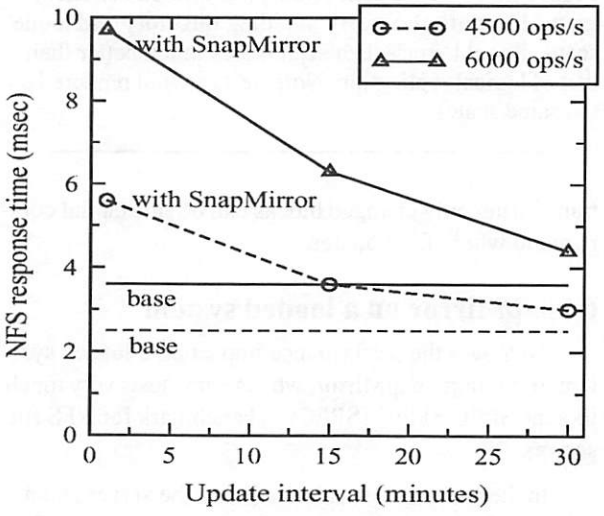


Figure 5. SnapMirror Update Interval vs. NFS response time. We measured the effect of SnapMirror on the NFS response time of SFS-like loads. By increasing SnapMirror update intervals, the penalty approaches a mere 22%.

run.

Even with the SnapMirror update period set to only one minute, the filer is able to sustain a high throughput of NFS operations. However, the extra CPU and disk load increases response time by a factor of two to over three depending on load.

Increasing the SnapMirror update period to 30 minutes decreases the impact on response time to only about 22% even when the system is heavily loaded with 6000 ops/sec. This reduction comes from two major effects. First, each SnapMirror update requires a new snapshot and a scan of the active map files. With less frequent updates, the impact of these fixed costs is spread over a much greater period. Second, as the update period increases, the amount of data that needs to be transferred to the destination per unit time decreases. Consequently SnapMirror reads as a percentage of the total load decreases.

7 Conclusion

Current techniques for disaster recovery offer data managers a stark choice. Waiting for a recovery from tape can cost time, millions of dollars, and, due to the age of the backup, can result in the loss of hours of data. Failover to a remote synchronous mirror solves these problems, but does so at a high cost in both server performance and networking infrastructure.

In this paper, we presented SnapMirror, an asynchronous mirroring package available on Network Appliance filers. SnapMirror periodically updates an online mirror. It provides the rapid recovery of synchronous remote mirroring but with greater flexibility and control in maintaining the mirror. With SnapMirror, data managers can choose to update the mirror at an interval of their choice. SnapMirror allows the user to strike the proper balance between data currency on one hand and performance and cost on the other.

By updating the mirror periodically, SnapMirror can transfer much less data than would a synchronous mirror. In this paper, we used traces of 12 production file systems to show that by updating the mirror every 15 minutes, instead of synchronously, SnapMirror can reduce data transfers by 30% to 80%, or 50% on average. Updating every hour reduces transfers an average of 58%. Daily updates reduce transfers by over 75%.

SnapMirror benefits from the WAFL file system's ability to take consistent snapshots both to ensure the consistency of the remote mirror and to identify changed blocks. It also uses the file system's active map to avoid transferring deallocated blocks. Trace analysis showed that this last optimization is critically important for no-overwrite file systems such as WAFL and LFS. Of the 12 traces analyzed, 10 would have seen no transfer reductions even with only update after 24 hours.

SnapMirror also leverages block level behavior to solve performance problems that challenge logical-level mirrors. In experiments comparing SnapMirror to dump-based logical-level asynchronous mirroring, we found that using block-level file system knowledge reduced the time to identify new or changed blocks by as much as two orders of magnitude. By avoiding a walk of directory and inode structures, SnapMirror was able to detect changed data significantly more quickly than the logical

schemes. Furthermore, transferring only changed blocks, rather than full files, reduced the data transfers by over 40%. Asynchronous mirror updates can run much more frequently when it takes a short time to identify blocks for transfer, and only the necessary blocks are updated. Thus, SnapMirror's use of file system knowledge at a block level greatly expands its utility.

SnapMirror fills the void between tape-based disaster recovery and synchronous remote mirroring. It demonstrates the benefit of combining block-level and logical-level mirroring techniques. It gives system administrators the flexibility they need to meet their varied data protection requirements at a reasonable cost.

8 Acknowledgments

The authors wish to thank Steve Gold, Norm Hutchinson, Guy Harris, Sean O'Malley and Lara Izlan for their generous contributions. We also wish to thank the reviewers and our shepherd, Roger Haskin, for their helpful suggestions.

9 References

[Chutani92] S. Chutani, et. al. *The Episode File System.* Proceedings of the Winter 1992 USENIX Conference, San Francisco, CA, January 1992. pp. 43-60.

[Baker91] M. Baker, J. Hartman, M. Kupfer, L. Shirriff, J. Ousterhout. *Measurements of a Distributed File System.* Proceedings of the 13th Symposium on Operating System Principles. October 1991. pp. 198-212.

[EMC] EMC Symmetrix® Remote Data Facility. http://www.emc.com/.

[Growchowski96] E.G. Grochowski, R.F. Hoyt. *Future Trends in Hard Disk Drives.* IEEE Transactions on Magnetics, V32. May 1996. pp. 1850-1854.

[Hitz94] D. Hitz, J. Lau, M.A. Malcolm. *File System Design* for *an NFS File Server Appliance.* Proceedings USENIX Winter 1994 Conference. pp. 235-246. http://www.netapp.com/tech_library/3002.html

[Hutchinson99] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, S. O'Malley. *Logical vs. Physical File System Backup.* Proceedings Third Symposium on Operating System Design and Implementation. February 1999.

[Kistler92] J.J. Kistler, M. Satyanarayanan. *Disconnected Operation in the Coda File System.* ACM Transactions on Computer Systems, 10(1). February 1992.

[Kistler93] J.J. Kistler, *Disconnected Operation in a Distributed File System.* Technical Report CMU-CS-93-156. School of Computer Science, Carnegie Mellon University, 1993. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda/Web/docs-coda.html

[McKusick84] M.K. McKusick, W.J. Joy, S.J. Leffler, R.S. Fabry. *A Fast File System for UNIX.* Transactions on Computer Systems 2,3. August 1984. pp. 181-197

[Ousterhout85] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, J.G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th Symposium on Operating Systems Principles* (SOSP), Orcas Island, WA, December, 1985, pp. 15-24.

[Rosenblum92] M. Rosenblum, J.K. Osterhout. *The Design and Implementation of a Log-structured File System.* ACM Transactions on Computer Systems, Vol.10, No.1 (Feb. 1992), pp. 26-52.

[SPEC97] The Standard Performance Evaluation Corporation. SPEC SFS97 Benchmark. http://www.spec.org/osg/sfs97/.

[Storage] Storage Computer Corporation Omniforce© software. http://www.storage.com/

[Tridgell96] A. Tridgell, P. Mackerras. *The rsync algorithm.* Department of Computer Science Australian National University. TR-CS-96-05.

Safety, Visibility, and Performance in a Wide-Area File System

Minkyong Kim, Landon P. Cox, and Brian D. Noble
Electrical Engineering and Computer Science
University of Michigan
{minkyong,lpcox,bnoble}@umich.edu

Abstract

As mobile clients travel, their costs to reach home filing services change, with serious performance implications. Current file systems mask these performance problems by reducing the *safety* of updates, their *visibility*, or both. This is the result of combining the propagation and notification of updates from clients to servers.

Fluid Replication separates these mechanisms. Client updates are shipped to nearby replicas, called *WayStations*, rather than remote servers, providing inexpensive safety. WayStations and servers periodically exchange knowledge of updates through *reconciliation*, providing a tight bound on the time until updates are visible. Reconciliation is non-blocking, and update contents are not propagated immediately; propagation is deferred to take advantage of the low incidence of sharing in file systems.

Our measurements of a Fluid Replication prototype show that update performance is completely independent of wide-area networking costs, at the expense of increased sharing costs. This places the costs of sharing on those who require it, preserving common case performance. Furthermore, the benefits of independent update outweigh the costs of sharing for a workload with substantial sharing. A trace-based simulation shows that a modest reconciliation interval of 15 seconds can eliminate 98% of all stale accesses. Furthermore, our traced clients could collectively expect availability of five nines, even with deferred propagation of updates.

1 Introduction

Mobile devices have become an indispensable part of the computing infrastructure. However, networking costs continue to render them second class citizens in a distributed file system. Limits imposed by networking costs are not new, but they are no longer due to "last mile" constraints. With the widespread deployment of broadband connectivity, mobile users often find themselves in a neighborhood of good network performance. Unfortunately, they must reach back across the wide area to interact with their file servers; the latency and congestion along such paths impose a substantial performance penalty.

To cope with increased costs, wide-area and mobile file systems employ two techniques to limit their use of the remote server. The first technique is *caching*. When workloads have good locality and infrequent sharing, caching can avoid most file fetches from the server. The second technique is *optimistic concurrency control*. Clients defer shipping updated files until time and resources permit. Ideally, most updates will be overwritten at the client, and need never be sent to the server.

Deferring updates improves performance, but harms safety and visibility. An update is *safe* if it survives the theft, loss, or destruction of the mobile client that created it. Safety requires that the contents of an update reside on at least one other host. An update is *visible* if every other client in the system knows it exists. Visibility requires that notification of an update reaches all replicas in the system.

Current file systems needlessly combine safety and visibility by propagating the contents of an update, implicitly notifying the destination that the update exists. *Fluid Replication* separates the concerns of safety and visibility through the addition of secondary replica sites. These sites, called *WayStations*, act as server replicas to nearby clients, servicing uncached reads and writes. Client writes are propagated to a WayStation immediately, providing safety. WayStations periodically *reconcile* their updates with the servers for which they act as replicas; this exchanges update notifications, but not the contents of those updates. Since reconciliation involves only meta-data, it can be done frequently, providing bounded visibility. Put another way, Fluid Replication aggressively writes updates back to the WayStation, but periodically invalidates updates between WayStation and server.

To maintain a simple consistency model, Fluid Replication provides *copy semantics*; each replica behaves as if a reconciliation actually copied all new versions accepted by the other replica. To support this, each WayStation retains file versions in *escrow* in the event that they are referenced at other replicas. Escrow also allows us to provide wait-free reconciliations with no additional mechanism.

We have built a prototype of Fluid Replication, and evaluated it with a variety of benchmarks. Updates are isolated from wide-area networking costs in the absence of sharing; clients pay wide-area costs only for accesses to shared files. A trace-based simulation of Fluid Replication shows that a modest reconciliation interval of fifteen seconds provides a stale access rate of only 0.01%, compared to a sharing rate of 0.6%. Space devoted to escrow storage is modest but bursty, with a high water mark of less than 10 MB. The impact of deferred propagation on availability is also small; our traced client population would expect to see one failed access per year, collectively.

2 Related Work

A number of projects have explored file systems for mobile, wide-area clients. Many of the ideas in Fluid Replication stem from the Coda file system. Coda provides high availability through disconnected operation [13] for clients and server replication [16] between well-connected servers. Fluid Replication is orthogonal to both; we do not expect clients to avoid disconnection or server failure. Coda also supports weakly-connected operation [18] to utilize low bandwidth between a client and its servers. During weakly-connected operation, Coda clients defer updates, and ship them later through a process called *trickle reintegration*. This sends both the knowledge of an update and its contents to the server. Since this is expensive, Coda defers reintegration in the hopes that some updates will be canceled by overwrites or deletions. This *aging window* is set to ten minutes to capture an acceptable fraction of possible optimizations, trading network bandwidth for more aggressive propagation.

Ficus [9, 23] shares Coda's goal of providing optimistic file access, but uses a peer-to-peer architecture. Each update is accepted by a single replica and asynchronously propagated to other sites, but no effort is made to ensure that update messages arrive. To propagate missed updates, Ficus provides copy semantics through the actual exchange of updates between two replicas. Updates are discovered by a disk scan at each replica, a heavyweight process. Reconciliation exchanges both knowledge and content of updates. Since this is expensive, it is intended only for well-connected peers, making visibility dependent on the mobility pattern of clients.

Bayou [26] provides optimistic concurrency control for database applications, where distributed updates are eventually committed by a primary replica. Deno [11] extends the Bayou model to provide distributed commit. These systems exchange full update sets whenever convenient. Unlike Ficus, they log updates; unlike Coda, updates are represented by-operation rather than by-value. This inherently combines notification of an update and its contents. However, since databases exhibit many fine-grained, partial updates, this may be less of an issue than for file systems. Exchanges between peers rely on client mobility and communication patterns. So, these systems can offer eventual visibility, but cannot bound the time required to do so.

OceanStore [15] advocates an architecture for global storage. Like Fluid Replication, it envisions storage supported by a loose confederation of independent servers. A primary focus of OceanStore is safely using untrusted nodes in the infrastructure. This is also an important problem for Fluid Replication, which we address elsewhere [20]. OceanStore's consistency mechanism, like that of Ficus and Bayou, is based on epidemic algorithms, and so cannot bound visibility. OceanStore represents updates by operation rather than by value, combining notification of updates with the shipment of their contents.

TACT [33] provides a middleware service to coordinate the consistency of data used by wide-area applications. Unlike Fluid Replication, it combines visibility and safety of updates. However, it does provide strong bounds on visibility. In addition to temporal bounds—which are offered by Fluid Replication—it can also bound the number of unseen writes or the *weight* of those writes. Combining Fluid Replication's separation of safety and visibility with TACT's mechanisms to trigger reconciliation can offer applications a tunable consistency mechanism with isolation from wide-area networking costs.

xFS is a file system designed for tightly coupled clusters of workstations [1]. However, early designs focused on the wide area [31]. In this model, clients were aggregated based on physical proximity, and served by a *consistency server*. On update, a client would retain the contents of the update—reducing safety—and notify the consistency server that the update occurred. Consistency between these second-level replicas and the main storage server was to be managed conservatively, imposing wide-area networking costs on normal operation.

Finally, JetFile [8] is motivated by the same concerns as the early xFS design, though it provides a number of additional features. It uses a central version server that provides serialization, but does not hold data; objects are stored primarily on the client that creates them, reducing safety. Clients locate files with scalable reliable multicast [7]. As in Ficus, invalidations are sent best-effort. However, in JetFile, the central version server broadcasts invalidations periodically to ensure bounded consistency; this period provides consistency equivalent to Fluid Replication. JetFile's most serious drawback is that it depends on ubiquitous IP multicast.

3 Design

This section presents the design of Fluid Replication with an eye towards separating the concerns of safety and visibility. Clients treat WayStations, wide-area replica sites, exactly as they would a server. Each WayStation reconciles its updates with the server periodically; reconciliation exchanges notification of updates, but not their contents. In order to provide copy semantics without actually copying during reconciliation, replicas hold reconciled updates in *escrow*. Due to the escrow mechanism, reconciliation does not require any node to hold a lock during a network round trip.

3.1 Context

Fluid Replication is designed to complement a client-server architecture, such as that of AFS [10]. In the AFS model, servers provide the centralized point of administration. They are expected to be carefully maintained to provide the best possible availability to their clients. Clients consist of workstations and mobile devices. Clients are considered the property of their users; they are not carefully administered and are much more likely to fail. This is particularly true of mobile clients.

The AFS model uses *callbacks* for consistency. Before using an uncached file, a client must first fetch it from the server. As a side effect of the fetch, the server establishes a callback on that file. The client can then use the now-cached copy. If a client modifies a file, the new version is sent to the server on `close`; this is called a *store* event. The server *breaks callback* on any other clients caching this file, invalidating their copies, before accepting the store. Unix semantics forbids invalidation of an open file; such files are invalidated after they are closed. If a file subject to pending invalidation is modified, the close returns an error.

3.2 WayStations: Hosting Replicas

The success of Fluid Replication depends on *WayStations*, which act as replicas for servers. They provide caching and immediate safety to nearby clients without penalizing performance, and act as reconciliation partners with remote servers. A WayStation is able to host replicas of any service for any client, regardless of administrative domain, and can host replicas of multiple servers concurrently.

In our architecture, WayStations play a role similar to that of servers; one model for a WayStation is as a fee-based service, much as broadband connectivity is for travelers today. In contrast to clients, they are carefully administered machines, and are relatively permanent members of the infrastructure. While they can be independently created, they are not expected to be transient. For example, a file server could also provide WayStation services to foreign clients. However, an end-user machine is not an appropriate WayStation because it may be turned off or disconnected at any time.

The decision of whether or not to use a WayStation is client-driven. Clients estimate the quality of the network between themselves and their current replica site [12]. If the network quality becomes poor, the client looks for a WayStation close enough to improve matters, and initiates a replica on it. Such searches are carried out in a neighborhood near the client through a process called *distance-based discovery* [19]. The WayStation informs the server of initiation. This notification is the point at which the invalidation semantics on a client change. It is an expensive operation, but it enables the server to drop any outstanding callbacks for this client. Requiring callback breaks for wide-area clients would substantially penalize local-area clients.

The WayStation fetches objects on demand. We considered prefetching, but rejected it. It is not clear that inferred prefetching [17] can provide requests in time to be useful given wide-area delays. We are also unwilling to require that applications disclose prefetch requests [24].

Foreign clients must trust WayStations to hold cached files securely and reconcile updates promptly. While one cannot prevent WayStation misbehavior completely, we plan to provide mechanisms to prevent exposure and reliably detect data modification or repudiation of accepted updates. This allows WayStation/client relationships to be governed by a contract. Parties to a contract are not prevented from breaching it; rather, the contract specifies penalties and remedies in the event of a breach. The details of our security architecture are beyond the scope of this paper [20]. We comment further on it in the context of future work in Section 6.

3.3 Reconciliation: Managing Replicas

Once a client has selected a WayStation, that WayStation is treated as the client's file server. Clients cache files from WayStations, which manage callbacks for those clients. Dirty files are written back to WayStations on `close`. WayStations maintain update logs using a mechanism similar to Coda's *client modify log* [13]. It contains file stores, directory operations, and meta-data updates. Redundant log records are removed via cancellation optimizations.

The server maintains an update log whenever one or more WayStations hold replicas of that server. The server also tracks which files are cached at the WayStation, called the *interest set*. The bookkeeping for interest sets is similar to that for callbacks, but the information is used only during reconciliation.

Periodically, each WayStation reconciles its update log with that of the server, exposing the updates made at

each replica to the other one. To initiate a reconciliation, the WayStation sends the server its update log plus a list of files the WayStation has evicted from its cache. The server removes the latter from the WayStation's interest set, and checks each WayStation log record to see if it is serializable. If it is, the server invalidates the modified object, and records the WayStation as the replica holding that version.

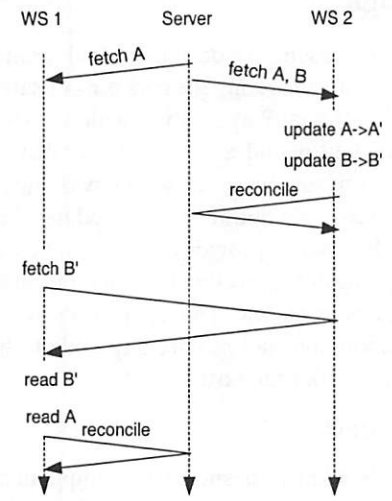
The server responds with a set of files to invalidate, and a set of files that are now in conflict, if any. The invalidation set consists of any updates accepted by the server—whether from a client or through another WayStation—that also reside in the WayStation's interest set. Since the WayStation will invalidate them, the server can remove them from the interest set as well.

WayStations truncate their update logs on successful reconciliation. However, since the server maintains a merged log of all replica sites, it cannot truncate its log immediately. The server maintains the last reconciliation time for each WayStation holding a replica; let t_{oldest} be the earliest such time. The server need only hold log records from t_{oldest} forward, since all older records have already been seen by all WayStations. Slow WayStations can force the server to keep an arbitrarily large log. This policy is in contrast to that taken by Coda's implementation of optimistic server replication. In Coda, only the tails of large update logs are retained; objects with discarded log records are marked in conflict and will need manual repair. Given Coda's presumption of tightly-coupled and jointly-administered replicas, this is an appropriate design choice. However, since WayStations are administered by independent entities, it would be unwise to allow a WayStation's absence to necessitate many manual repairs.

If each WayStation reconciles at a constant rate, all updates are globally visible within twice the longest reconciliation period. The first reconciliation invalidates the server copy, and all other WayStation copies are invalidated during the next round. In order to provide clients with well-bounded visibility, reconciliation must be a lightweight operation. This is why reconciliations exchange only notification of updates, not their contents. Because sharing is rare, aggressively exchanging file contents increases reconciliation time without improving client access time. Leaving shared accesses to pay the full cost of wide-area network delays preserves performance, safety, and visibility for common-case operations.

3.4 Escrow: Providing Copy Semantics

When a replica site needs the contents of an invalidated file, what version does it obtain? The simplest approach would be to provide the most recent version at the time



This figure shows how fetching the most recent version of a missing file can result in the propagation of out-of-order updates. WayStation 2 updates A then B. WayStation 1 sees the update to B first, because A had already been cached without an intervening reconciliation.

Figure 1: Propagating Updates Out-Of-Order

it is requested; we call this *last-version semantics*. Unfortunately, last-version semantics allow WayStations to see updates out of order. Instead, Fluid Replication's copy semantics guarantees that updates are seen in the order they are made. We believe this is important, as other studies have shown that users do not always understand the implications of complicated consistency mechanisms [6].

Figure 1 illustrates how out-of-order updates can arise under last-version semantics. There are two files, A and B, shared by clients at two WayStations, WS_1 and WS_2. Assume that WS_1 caches A, and WS_2 caches both A and B. Suppose that a client at WS_2 updates A then B, and then reconciles. The server now knows about both new versions held at WS_2. If a client at WS_1 were to reference both files A and B without an intervening reconciliation, it would see the old version of A but the new version of B. When WS_1 eventually reconciles, the update of A would be known, but it would have been seen out of order.

One solution would be to require clients to fetch all updated objects of interest whenever fetching any. Fluid Replication eschews this approach, as it would result in more work across the wide-area path between WayStation and server. Instead, we appeal to copy semantics. Under copy semantics, WS_1 would have read B rather than B', because B was the current version when WS_1 last reconciled. This is despite the fact that B was not in WS_1's cache and the update, B', had already been reconciled by WS_2. Copy semantics guarantees that each

WayStation see a complete prefix of all activity at each WayStation, and therefore provides a consistent view of the system. It is important to note that providing copy semantics does not group related updates into atomic actions. In other words, copy semantics cannot prevent seeing one update but not a related one. However, all such mis-matches will be explicable in reference to "real time". In Bayou's terminology, this provides *monotonic writes* [30].

To provide copy semantics to a WayStation requesting a file, Fluid Replication must supply the version known to the server at the WayStation's last reconciliation. This version may reside on the server directly, or it may reside on the WayStation that had accepted it. Therefore, a replica site that makes an update visible to another via reconciliation must retain a copy of the update for as long as the other replica might refer to it. We say that such copies are held in *escrow*.

If a client at the server refers to a version escrowed at some WayStation, the server *back-fetches* it; responsibility for the escrowed version passes from WayStation to server. If a client at another WayStation references it, the fetch request is first sent to the server. The server obtains it from the escrowing WayStation—back-fetching as before—and then forwards it on to the requesting WayStation. Responsibility for escrow always migrates from a WayStation to the server, never the reverse.

For escrow to be practical, we must prevent unbounded storage growth. To see how this might occur, consider a WayStation at which a file is updated between each reconciliation. Without knowing which versions are visible to other replica sites, the WayStation would be forced to hold all of them. The key to managing this growth is to observe that only the *most recent* version visible to each replica need be held in escrow. Any other versions are said to be *irrelevant*, and can be safely pruned.

A simple way to prune versions is to track which are old enough to be globally irrelevant. Recall that t_{oldest} is the latest time after which all WayStations have reconciled successfully. Sending this time to a WayStation during a reconciliation allows the WayStation to prune old versions. Let $f_1 \ldots f_n$ be the sequence of updates to some file f at a WayStation that were reconciled with the server. Let f_i be the most recent of those updates performed before t_{oldest}. Clearly, all versions $f_1 \ldots f_{i-1}$ are irrelevant; each WayStation would request either f_i or some later version.

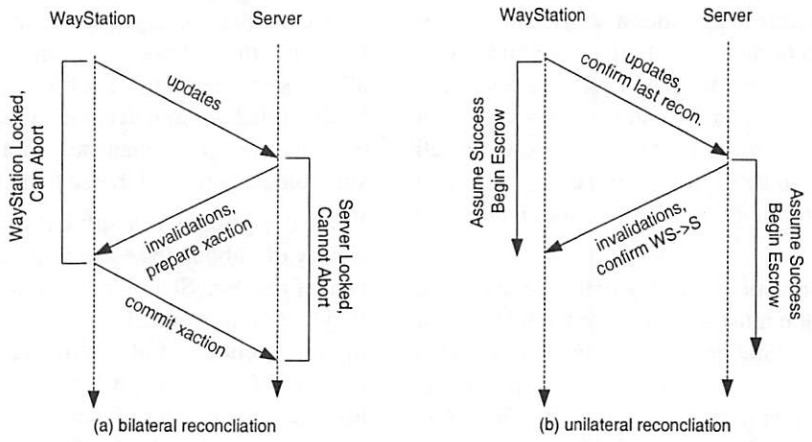
If all WayStations reconcile at the same rate, each WayStation must keep at most one additional version of a file in escrow. This is because each new reconciliation would find that t_{oldest} had advanced past the time of that WayStation's last reconciliation. A version enters escrow only when a more recent version is created; after reconciliation, t_{oldest} advances past the new version, rendering the old one irrelevant. Note that this scheme allows some versions of a file stored at WayStations to be discarded without ever being sent to the server. However, the t_{oldest} mechanism guarantees that no replicas still hold an active reference to that version.

While this pruning is helpful, it does not prevent the possibility of unbounded escrow space. Consider what happens if one WayStation is slow to reconcile. This single WayStation prevents the advancement of t_{oldest}, requiring the retention of all future versions at each WayStation. In effect, a single WayStation can hold all others hostage. So, instead of sending only t_{oldest}, the server sends a sorted list of timestamps $t_{oldest} \ldots t_{newest}$, where each entry lists the time of last reconciliation by some WayStation. If two or more visible versions lie between adjacent timestamps, only the last one is needed. Under this scheme, a slow-to-reconcile WayStation requires only one additional copy in escrow.

The escrow mechanism depends on the fact that WayStations are closer in spirit to servers than they are to clients. Because they hold versions in escrow for other replicas, they must be reasonably available. An alternative design for Fluid Replication allows a client to act as its own WayStation. While this approach would be simpler, it can leave escrowed copies unavailable to other sites. End-user machines are often unavailable for days at a time, and may even disappear completely [4].

Under escrow, file updates can reside on WayStations indefinitely. One could argue that this is proper; if an update is never needed elsewhere, it should never be shipped from the WayStation. However, there are two reasons to ship versions in advance of need. First, administrative tasks such as backup are greatly simplified if files eventually migrate to the server. Second, migration limits the risk of unavailable escrowed copies and data loss. Therefore, any version that has not been updated for one hour is sent to the server. We chose one hour to take advantage of most cancellation opportunities [18], while balancing propagation overhead with the need for availability.

It is important to note that copy semantics cannot guarantee in-order updates if those updates are destined for more than one server. This is because there is no common synchronization point. A WayStation could provide this guarantee through synchronous, atomic reconciliations across all of its servers. However, such steps introduce wide-area latencies into the critical path of operations, counter to Fluid Replication's philosophy. Alternatively, the servers could synchronize amongst themselves, but with similar drawbacks. We expect that most users will have one "home" file system, obtaining the



This figure illustrates the difference between bilateral and unilateral reconciliations. Bilateral reconciliations require three messages. Each replica must be locked during an expensive round trip, and WayStation failures can hold the server hostage. In contrast, unilateral reconciliations optimistically assume that exposures are successful, and begin escrow immediately. They require two messages rather than three, employ only short-term, local locks, and hold no replicas hostage to failure.

Figure 2: The Advantage of Unilateral Reconciliations

benefit of in-order updates while paying only the small cost of local escrow.

3.5 Fault Tolerance in Reconciliation

The simplest model for reconciliation is *bilateral*: the atomic exchange of all update log records between a WayStation and a server. Unfortunately, this simple model is problematic in the face of node or network failures. Atomic exchanges require a two-phase commit protocol. One node must *prepare* the transaction, agreeing to either commit or abort until the other party confirms the result. In the meantime, the prepared node must block many operations until the transaction completes.

The difficulty is caused by store operations during a bilateral reconciliation. These stores cannot be serialized before the reconciliation. Doing so would require that they had been in the reconciled update log, which is impossible. The stores cannot be serialized after the reconciliation either, since they may refer to a file that the reconciliation will invalidate. Therefore, store operations issued during a bilateral reconciliation must block until it completes. In the presence of failures, stores may be blocked indefinitely. Put another way, bilateral reconciliation imposes wide-area networking costs on clients even in the absence of sharing; this runs counter to Fluid Replication's philosophy.

In light of these problems, we split a single, bilateral reconciliation into two *unilateral* ones. These alternatives are illustrated in Figure 2. WayStations initiate reconciliations, as before. However, as soon as the reconciliation message is sent, the WayStation assumes it will be received successfully. It can continue processing client requests immediately, placing versions in escrow as needed. The server likewise assumes success and begins escrow after sending its message. As a side effect, the server's message confirms the exposures assumed by the WayStation. The next WayStation request confirms the completion of the prior reconciliation.

There are several benefits to this approach. WayStations block updates only while computing a reconciliation message; no locks are held across expensive round trips. This appeals to the common case by placing potentially-visible versions in escrow immediately, rather than waiting for confirmation. Since the escrow mechanism is needed to provide copy semantics, no additional complexity is required. Finally, the third reconciliation message—required to keep bilateral locking times short—is implied by future messages. This *piggybacking* is similar to the process Coda uses to manage updates to well-connected, replicated servers [27]. Unlike Coda's COP2 messages, confirmation of reconciliations can be deferred indefinitely. The only penalty of an undetected failure is a larger escrow.

Unilateral reconciliations provide a good approximation to the desired effect of bilateral ones. The server sees a single, atomic event, while WayStations are able to allow clients to continue during the wide-area operation. In addition to a potential increase in escrow size, there is a slight widening of the conflict window, because concurrent writes are now allowed. Suppose that a WaySta-

tion initiates a reconciliation, then accepts an update to file f that is later invalidated by the server. This causes f to be in conflict. With bilateral reconciliation, the update to f would have been delayed until after invalidation and then rejected, avoiding a conflict. However, given the low incidence of write-shared files—particularly over such short time frames—it is unlikely that such spurious conflicts will occur in practice.

4 Implementation

Our Fluid Replication prototype consists of three components: a client cache manager, a server, and a WayStation. Each of these components is written primarily in Java. There are several reasons for this, foremost among them Java's clean combination of thread and remote procedure call abstractions and the benefits of writing code in a type-safe language.

The bulk of the Fluid Replication client is implemented as a user-level cache manager, supported by a small, in-kernel component called the *MiniCache* [29]. The MiniCache implements the vnode interface [14] for Fluid Replication. It services the most common operations for performance, and forwards file operations that it cannot satisfy to the user-level cache manager.

Calls are forwarded from the kernel to the cache manager across the Java Native Interface, JNI. The calls are then satisfied by one of a set of worker threads, either from the local disk cache or via Remote Method Invocation, RMI, to the appropriate replica. Fluid Replication uses write-back caching; a close on a dirty file completes in parallel with the store to the replica. The client supports dynamic rebinding of servers and WayStations for migration; currently, our prototype migrates only on user request.

WayStations and servers share much of the same code base, since their functionality overlaps. Data updates are written directly to the replica's local file system. Metadata is stored in memory, but is kept persistently. Updates and reconciliations are transactional, but we have not yet implemented the crash recovery code. We use Ivory [2] to provide transactional persistence in the Java heap; used in this way, it is similar to RVM [28].

The decision to write the client in Java cost us some performance, and we took several steps to regain ground. Our first major optimization was to hand-serialize RMI messages and Ivory commit records. The default RMI skeleton and stub generator produced inefficient serialization code, which we replaced with our own. This reduced the cost of a typical RMI message by 30%.

The second major optimization concerned the crossing of the C-Java boundary. Each method call across this boundary copies the method arguments onto the Java stack, and returned objects must be copied off of the Java stack. We were able to avoid making these copies by using *preserialized objects*, provided by the Jaguar package [32]. Jaguar allows objects to be created outside the Java VM, and still be visible from within. We used this to pass objects, copy-free, between our C and Java code.

5 Evaluation

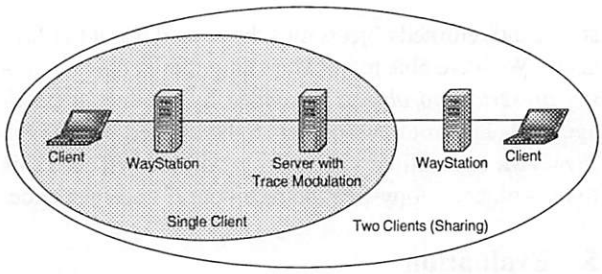
In evaluating Fluid Replication, we set out to answer the following questions:

- Can Fluid Replication insulate clients from wide-area networking costs?
- What is the impact of sharing on performance?
- How expensive is reconciliation?
- Can Fluid Replication provide the consistency expected by local-area clients to those in the wide area?
- How does escrow affect availability?
- What are the storage costs of escrow?

These questions concern the performance seen by individual clients and the behavior of the system as a whole. To measure client performance, we subjected our prototype to a set of controlled benchmarks. We explored system behavior through the use of trace-based simulation.

Our benchmarks ran on the testbed depicted in Figure 3. The WayStations are connected to the server via a *trace modulated* network. Trace modulation performs application-transparent emulation of a slower target network over a LAN [22]. We have created modulation traces that emulate the performance of a variety of different wide-area networking scenarios, listed in Table 1. Latency numbers in these traces are in *addition* to baseline latency, while bandwidth specifies the bottleneck capacity. All testbed machines run the Linux 2.2.10 kernel. The server and WayStations have 550 MHz Pentium III Xeon Processors, 256 MB of RAM, and 10K RPM SCSI Ultra Wide disks. The clients are IBM ThinkPad 570s; these machines have 366 MHz mobile Pentium IIs with 128 MB of memory.

For the trace-based studies, we collected traces comprising all activity on a production NFS server over one week. This server holds 188 users' home directories, plus various collections of shared data, occupying 48 GB. The users are graduate students, faculty, and staff spread throughout our department. They come from a variety of research and instructional groups, and have diverse storage needs. Generally speaking, the clients are not mobile, so they may not be wholly representative of our target domain. However, prior studies suggest that, at the operation level captured by our traces, mobile and desktop behavior are remarkably similar [21].



This figure illustrates our benchmarking topology. Each client is well connected to its WayStation, but traffic between a WayStation and the server is subject to trace modulation.

Figure 3: Benchmark Topology

Scenario	Latency (ms)	Bandwidth
local area	0.0	10 Mb/s
small distance	15.0	4 Mb/s
medium distance	33.5	3 Mb/s
large distance	45.0	1 Mb/s
intercontinental	55.0	1 Mb/s
low bandwidth	0.0	56 Kb/s
high latency	200.0	10 Mb/s

This table lists the parameters used in each of our trace modulation scenarios. The local area scenario is the baseline against which we compare. The next four were obtained by measuring small `ping` and large `ftp` performance to four different sites. Bandwidth numbers are increased over `ftp` throughput by 20% to account for the difference in metric. The last two are used only to determine trends as latency or bandwidth worsen, and modify parameters orthogonally. Latencies are one-way; bandwidths are symmetric.

Table 1: Trace Modulation Parameters

Traces were collected using `tcpdump` on the network segment to which the server was attached. These packet observations were then fed into the `nfstrace` tool [3], which distilled the traces into individual fetch and store operations. Note that this tool does not record operations satisfied by a client's cache. However, since NFS clients do not use disk caches, this will overstate the amount of read traffic a Fluid Replication client would generate. For the purposes of our analyses, we assume that each client host resides on a separate WayStation. The traces name 84 different machines, executing 7,980 read operations and 16,977 write operations. There are relatively few operations because most of our client population did not materially contribute to the total. Seven hosts account for 90% of all requests, and 19 hosts account for 99% of all requests.

5.1 Wide-Area Client Performance

How effectively does Fluid Replication isolate clients from wide-area networking costs? To answer this question, we compare the performance of Coda, AFS, and Fluid Replication in a variety of networking conditions. For Coda, we ran Coda 5.3.13 at the server and the client. For AFS, we used OpenAFS, a descendant of AFS 3.6, as the server, and Arla 0.35.3 as the client. To provide a fair comparison, we set up our Coda volume on a single server, rather than a replicated set.

Our benchmark is identical to the Andrew Benchmark [10] in form; the only difference is that we use the *gnuchess* source tree rather than the original source tree. Gnuchess is 483 KB in size; when compiled, the total tree occupies 866 KB. We pre-configure the source tree for the benchmark, since the configuration step does not involve appreciable traffic in the test file system. Since the Andrew Benchmark is not I/O-bound, it will tend to understate the difference between alternatives. In the face of this understatement, Fluid Replication still outperforms the alternatives substantially across wide-area networks.

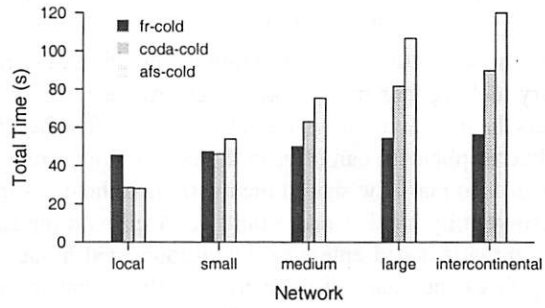
We tested each file system with both cold and warm caches. In the case of AFS and Coda, a "warm cache" means that the clients already hold valid copies of the *gnuchess* source tree. In the case of Fluid Replication, the source tree is cached on the WayStation.

Figures 4 compares the total running times of the Fluid Replication, Coda, and AFS clients under different network environments. Figure 4(a) gives the performance with a cold cache, and Figure 4(b) shows it with a warm cache. Each experiment comprises five trials, and the standard deviations are less than 2% of the mean in all cases.

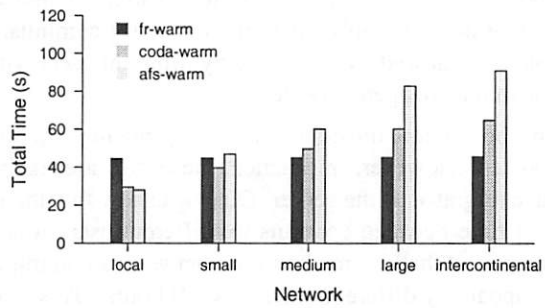
With a cold cache and a well-connected server, Coda and AFS outperform Fluid Replication. We believe this is due to our choice of Java as an implementation language and the general maturity level of our code. We see no fundamental reason why Fluid Replication's performance could not equal Coda's. They have identical client architectures, and the client/WayStation interactions in Fluid Replication are similar to those between client and server in Coda.

As the network conditions degrade, the cold-cache times of Coda and AFS rapidly increase while those of Fluid Replication increase slowly. All systems must fetch source tree objects from the server, and should pay similar costs to do so. The divergence is due to the systems' different handling of updates. In Fluid Replication, all updates go only to the nearby WayStation. In Coda and AFS, however, updates must go across the wide area to the server.

With a warm cache, the total running time of Fluid Replication remains nearly constant across all network environments. This is because the updates are propagated



(a) cold cache



(b) warm cache

This figure compares the total running times of Coda, AFS and Fluid Replication under a variety of network environments. The upper figure gives the results for a cold cache, and the lower figure shows them for a warm cache. With a cold cache, Fluid Replication is least affected by network costs among three systems. With a warm cache, the performance of Fluid Replication does not appreciably degrade as network costs increase.

Figure 4: Client Performance

only to the WayStation; invalidations are sent to the server asynchronously. The running time of AFS and Coda increases as the network degrades. They must propagate updates to the server during the *copy* phase and write object files back to the server during the *make* phase. The Coda client never entered weakly-connected mode, since the bandwidths in our sample traces were well above its threshold of 50 KB/s. Had Coda entered weakly-connected mode, its performance would be identical to Fluid Replication's, but its updates would be neither safe nor visible for many minutes.

Table 2 shows the normalized running time of Fluid Replication, with the local-area case serving as the baseline. While the running time increases by as much as 24.1% when the cache is cold, it remains nearly constant when the cache is warm. Looking only at the four scenarios generated by `ping` and `ftp` experiments, there appears to be a slight growth trend, but it is within observed variance. Only the results for Small and Intercontinental are not identical under the t-test; all other pairs are statistically indistinguishable. To conclusively rule out a true trend, we also ran the warm-cache Fluid Replication experiment across two more-demanding networking scenarios. The first decreased bandwidths to 56 Kb/s, but added no additional latency. The second increased one-way latency to 200 ms, but placed no additional bandwidth constraints. In both cases, Fluid Replication's running time was less than that for the Intercontinental trace. Therefore, we conclude that that Fluid Replication's update performance does not depend on wide-area connectivity. Of course, this is due to deferring the work of update propagation and the final reconciliation — neither of which contribute to client-perceived performance. Sections 5.4 and 5.5 quantify any reduction in consistency or availability due to deferring work.

Trace	Cold	Warm
Small	1.040	1.007
Medium	1.103	1.012
Large	1.200	1.013
Intercontinental	1.241	1.020
Low Bandwidth		1.014
High Latency		1.007

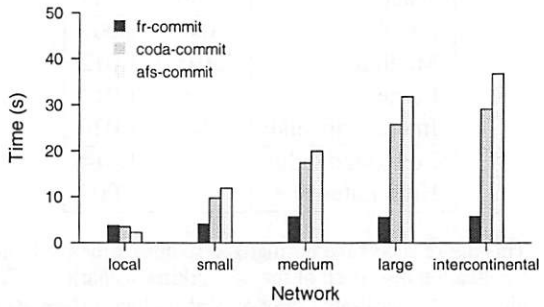
This figure shows the normalized running time for Fluid Replication over each of the networking scenarios in Table 1. The local-area case served as the performance baseline in each case. When the WayStation cache is cold, performance decreases as wide-area networking costs increase. However, no consistent trend exists when the WayStation cache is warm.

Table 2: Fluid Replication over Wide-Area Networks

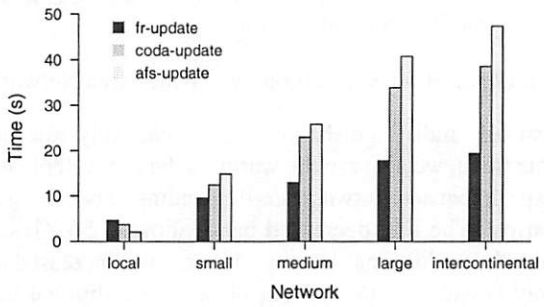
5.2 Costs of Sharing

Our next task is to assess the potential impact that deferred propagation has on sharing between wide-area clients. We devised a benchmark involving two clients sharing source code through a CVS repository. In the benchmark, clients C1 and C2 are attached to WayStations W1 and W2, respectively. In the first phase, C1 and C2 each check out a copy of the source tree from the repository. After changing several files, C1 commits those changes to the repository. Finally, C2 updates its source tree. Both the repository and working copies reside in the distributed file system. When the benchmark begins, both clients have the repository cached.

We used the Fluid Replication source tree for this benchmark. The changes made by C1 are the updates made to our source tree during the four of the busiest days recorded in our CVS log. At the beginning of the trial, the source tree consists of 333 files totaling 1.91 MB. The committed changes add four new files, totaling seven KB, and modify 13 others, which total 246 KB. We reran this activity over the two-client topology of Figure 3.



(a) commit



(b) update

Our sharing benchmark replayed sharing from the CVS log of our Fluid Replication source tree. The benchmark consisted of five phases: a checkout at C1, a checkout at C2, an edit at C1, a commit at C1, and an update at C2. This figure shows the time to complete the commit and update phases for AFS, Coda, and Fluid Replication.

Figure 5: Sharing Benchmark

Figure 5 shows our results for the commit and update phases, run over Fluid Replication, AFS, and Coda; each bar is the average of five trials. We do not report the results of the checkout and edit phases, because they exhibit very little sharing.

Unsurprisingly, the cost of committing to the repository was greater for AFS and Coda than for Fluid Replication. This is because Fluid Replication clients ship updates to a nearby WayStation. AFS and Coda must ship data and break callbacks over the the wide area.

What is surprising is that the update phase is also more costly for AFS and Coda. One would think that Fluid Replication would perform poorly, since file data has to traverse two wide area paths: from the first WayStation to the server, and then to the second WayStation. The unexpected cost incurred by AFS and Coda stems from the creation of temporary files. The latter are used to lock the repository and back up repository files in case of failure. WayStations are able to absorb these updates, and later optimize them away, since they are all subject to cancellation. AFS and Coda clients, on the other hand, send every update to the server. Furthermore, these updates break callbacks on the far client.

It is important to note that hiding the creation of temporary locking files may improve performance, but it renders the intended safety guarantees useless. The best that Fluid Replication can offer in the face of concurrent updates is to mark the shared file in conflict; the users must resolve it by hand. Coda would be faced with the same dilemma if it had entered weakly-connected mode during this benchmark. However, we believe that in practice, optimism is warranted. Even in the case of CVS repositories, true concurrent updates are rare. Our own logs show that commits by different users within one minute occurred only once in over 2,100 commits. A commit followed by an update by different users within one minute happened twice.

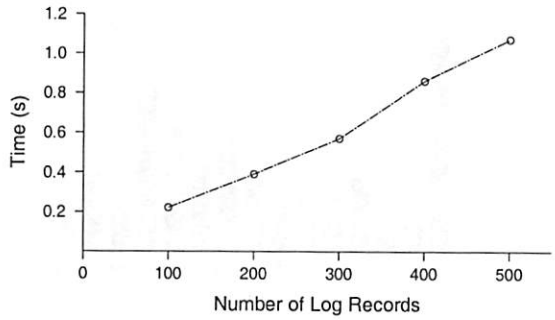
This benchmark illustrates the cost of obtaining deferred updates. However, in practice, these files are likely to have migrated to the server. Our logs show that the median time between commits by different users was 2.9 hours, and that the median time between a commit and an update by different users was 1.9 hours. This would provide ample opportunity for a WayStation to asynchronously propagate shared data back to the server before it is needed.

5.3 Reconciliation Costs

To be successful, Fluid Replication must impose only modest reconciliation costs. If reconciliations are expensive, WayStations would be able to reconcile only infrequently, and servers could support only a handful of WayStations. To quantify these costs, we measured reconciliations, varying the number of log records from 100 to 500. To put these sizes in context, our modified Andrew Benchmark reconciled as many as 148 log records in a single reconciliation.

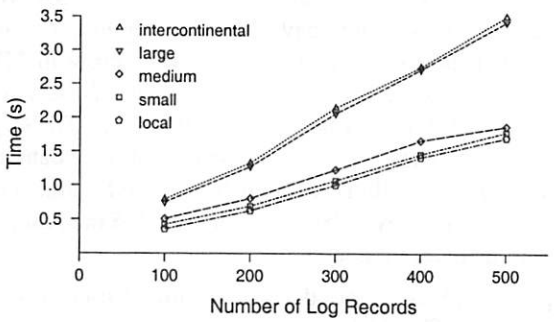
Figure 6 shows the reconciliation time spent at the server as the number of log records varies; each point is the average of five trials. This time determines the number of WayStations a server can handle in the worst case. Server-side time increases to just under 1.1 seconds for 500 log records. In practice we expect the costs to be much smaller. The week-long NFS trace would never have generated a reconciliation with more than 64 log records.

Figure 7 shows the total time for a reconciliation measured at a WayStation as the number of log records varies. This includes the server-side time, RMI overheads, and network costs. The total reconciliation times for the local area, small distance, and medium distance traces do not vary significantly. This means that, at these speeds, bandwidth is not the limiting factor. Profiling of the reconciliation process suggests that RMI—even with



This figure shows the reconciliation times spent at a server as the number of log records varies. These times determine the number of WayStations a server can handle.

Figure 6: Reconciliation Time at Sever



This figure shows the total reconciliation times in seconds as the number of log records varies. These times determine the upper limit on how frequently a WayStation can reconcile with the server. For all sizes, they are much shorter than our default reconciliation period of 15 seconds.

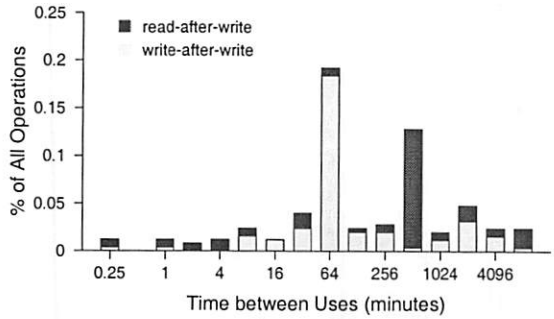
Figure 7: Reconciliation Time at WayStation

our hand-optimized signatures—is the rate-limiting step. However, at 1 Mb/s, the bandwidth of the large distance and intercontinental traces, the bottleneck shifts to networking costs. In any event, the reconciliation times are much shorter than our default reconciliation period of 15 seconds, allowing the WayStation to reconcile more frequently if sharing patterns warrant.

5.4 Consistency

The consistency offered by Fluid Replication depends on two factors: the frequency of reconciliation and the time between uses of a shared object. Section 5.3 quantified the former. In this section, we address the incidence of sharing observed in a real workload.

To determine how often sharing happens and the time between shared references, we examined our week-long NFS client traces. For this analysis, we removed references to user mail spools from our traces. A popular mail



This figure shows the percentage of operations that caused sharing. The x-axis shows the time between uses of a shared objects in minutes, and the y-axis shows the percentage of total operations that exhibited sharing. The top part of bar shows the read-after-write sharing; the bottom part shows the write-after-write. Only 0.01% of all operations caused sharing within 15 seconds. Just over 0.6% of all operations exhibited any form of sharing.

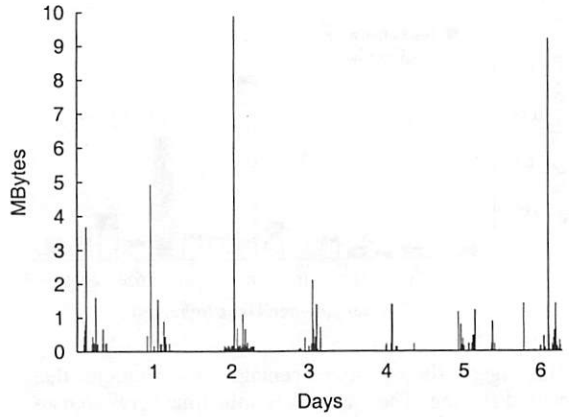
Figure 8: Sharing

client in our environment uses NFS rather than IMAP for mail manipulation. Many users run these clients on more than one machine, despite NFS's lack of consistency guarantees [25], generating spurious shared references. Since we would expect mobile users to use IMAP instead, we excluded these references.

Figure 8 shows the percentage of all references to objects written previously at another replica site. The top part of each bar shows read-after-write sharing, and the bottom part shows write-after-write. As expected, sharing is not common, especially over short periods of time. Only 0.01% of all operations caused sharing within 15 seconds. The total fraction of references that exhibited sharing during the week was just over 0.6% of all operations. Note that these numbers are pessimistic, as we have assumed that each client uses a distinct WayStation. The graph shows some interesting periodic behavior; unfortunately, with network-level traces, we are unable to identify the processes causing it.

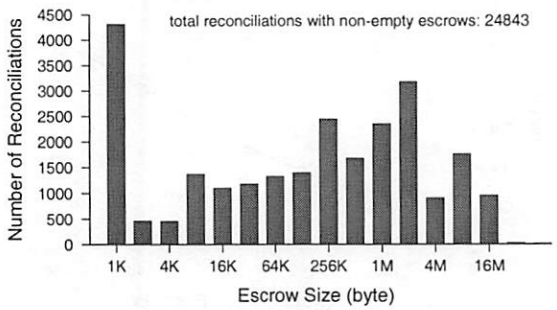
5.5 Availability

Because WayStations do not propagate file contents to the server immediately, a failed WayStation could keep a client from retrieving a needed update. To gauge how likely such a scenario might be, we fed our NFS traces into a Fluid Replication simulator. We augmented the trace with WayStation reconciliations and failure/recovery pairs. Reconciliations were scheduled every 15 seconds from the WayStation's first appearance in the trace. We assume a mean time to failure of 30 days and a mean time to repair of one hour, both exponentially distributed. These parameters were chosen to rep-



This figure shows the size of global escrow. The x-axis shows time in days from the beginning of the NFS trace, and the y-axis shows size of escrow.

Figure 9: Global Escrow Size



This figure shows the distribution of non-empty escrow sizes. The x-axis shows time in days from the beginning of the NFS trace, and the y-axis shows the number of reconciliations resulting in an escrow of that size or less.

Figure 10: Non-Empty Escrow Distribution

resent typical uptimes for carefully administered server machines; they are the same as those used in the xFS study [5]. Note that we did not model server failures; our intent is to explore Fluid Replication's *additional* contribution to failures.

For each trial of the simulation, we varied the random seed controlling failures and repairs. We then counted the number of back-fetches that fail due to the unavailability of a WayStation. We took over five million trials of the experiment in order to provide reasonable confidence intervals for a very small mean.

Recall that during our trace there were 24,957 total requests made of the WayStations. Of those, 243 requests required a back-fetch from one WayStation to another. We first simulated Fluid Replication when updates that had not been overwritten or back-fetched were propagated by the WayStation after 1 hour. Across five million trials, 187,376 requests failed, for an average of 1.46×10^{-6} failures per operation, with a 90% confidence interval of $\pm 8.83 \times 10^{-7}$; this is equivalent to five nines' availability. Expressed in time, we observed an average of 0.037 failures per week, or roughly one failed access every 27 weeks for our client population.

5.6 Costs of Escrow

Our final question concerns the storage costs of escrow. To determine them, we fed the week-long NFS traces into our simulator. Each WayStation in the trace reconciles every 15 seconds from the moment it is first named, as before. After an update has been reconciled, it is marked eligible for escrow. Any subsequent write to the file checks for this mark, and, in its presence, preserves the old version in case it is needed. Updates are removed from escrow as described in Section 3.4.

Figure 9 shows how global escrow size changes with time. The x-axis plots days from the beginning of the trace and the y-axis shows the size of escrow in MB. Global escrow size was almost always zero, and never exceeded 10 MB over the course of the week. The large variance matches the burstiness expected of update activity. Over the lifetime of our trace, WayStations collectively handle over 280 MB; compared to this, escrow requirements are modest.

Figure 10 gives a more detailed picture of escrow sizes over time. This histogram plots reconciliations resulting in non-empty escrows, showing the frequency with which each size was seen. Almost 89% of all non-empty escrows are 2MB or smaller.

6 Future Work

There are three main tasks ahead of us. First, we plan to use network estimation [12] and distance based discovery [19] to automate the WayStation migration and location processes. We have developed a technique that estimates network performance along the path between a client and a server using observations of request/response traffic. This estimator, borrowing techniques from statistical process control, follows underlying shifts in network performance while filtering out observed noise. In effect, the estimator adapts its behavior to prevailing networking conditions, selecting for agility or stability as appropriate. This estimator, combined with a cost-benefit analysis of WayStation migration, guides the search for a new replica site when the client moves too far from the old one.

Our second task is to address the need for trust between clients and WayStations [20]. Before a client will agree to use a WayStation, it must be assured of the privacy and integrity of cached data, and of non-repudiation of updates. Preventing exposure through encryption is

straightforward, though managing keys can be subtle. It is also easy to reliably detect unauthorized modifications using cryptographic hashes. However, guaranteeing that a WayStation will correctly forward updates is difficult, if not impossible.

Rather than attempt to prove a WayStation trustworthy a priori, we plan to provide a *receipt* mechanism that enables a client to prove that a WayStation did not properly forward an update. When a client ships a version of a file to the WayStation, it receives a cryptographically-signed receipt for that update. The client can later check whether that version was properly reconciled; lack of reconciliation provides evidence of update repudiation. Receipts can be optimized away much as updates can, but clients must retain the authority to apply such optimizations. Furthermore, clients retain the latest version of any updated file until that version is known to either reside on the server or have been invalidated; the space required to do so is modest [13]. This guarantees that the latest version of a file is known to reside on either a client or the server, in addition to any untrusted WayStation copies. Thus, a WayStation that disappears from the system cannot take with it the only current version of a file, though earlier, escrowed versions may vanish.

Our third goal is to gain more experience with Fluid Replication and its use. While the NFS traces are informative, they do not capture precisely how users will use Fluid Replication. We plan to deploy a server in our department for day-to-day storage requirements, and provide users with WayStations and wireless gateways at home. This will allow client laptops to seamlessly migrate between locales, giving us valuable insights into the system's use.

7 Conclusion

As mobile clients travel, their costs to reach back to home filing services change. To mask these performance problems, current file systems reduce either *safety*, *visibility*, or both. This is a result of conflating safety and visibility into a single mechanism. They have different requirements, and so should be provided through different mechanisms.

Fluid Replication separates the concerns of safety and visibility. While traveling, a mobile client associates itself with a nearby *WayStation* that provides short-term replication services for the client's home file system. Updates are sent to the nearby WayStation for safety, while WayStations and servers frequently exchange knowledge of updates through *reconciliation* to provide visibility. Reconciliation is inexpensive and wait-free. WayStations retain copies of advertised updates in *escrow* until they are irrelevant to other replica sites.

An analysis of traffic in a production NFS server validates our design decisions. A modest reconciliation interval of fifteen seconds limits the rate of stale reads or conflicting writes to 0.01%. Reserving 10 MB for escrow space—a small fraction of disk capacity—is sufficient in the worst case. Measurements of a Fluid Replication prototype show that it isolates clients from most wide-area networking costs. Update traffic is not affected at bandwidths as low as 56 Kb/s or latencies as high as 200 ms. These gains offset increased sharing costs, even for workloads with substantial degrees of sharing. Escrow space requirements are modest: at its peak, it is less than 5% of the total data cached at WayStations. Despite the fact that update propagation is deferred, availability does not suffer. Our traced clients could expect five nines' availability.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] G. C. Berry, J. S. Chase, G. A. Cohen, L. P. Cox, and A. Vahdat. Toward automatic state management for dynamic web services. In *Network Storage Symposium*, Seattle, WA, October 1999.

[3] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the Winter 1992 USENIX Conference*, pages 333–343, Berkeley, CA, January 1992.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *2000 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.

[5] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160, Nashville, TN, May 1994.

[6] M. R. Ebling. *Translucent Cache Management for Mobile Computing*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1998.

[7] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[8] B. Gronvall, A. Westernlund, and S. Pink. The design of a multicast-based distributed file system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 251–264, New Orleans, LA, February 1999.

[9] J. S. Heidemann, T. W. Page, R. G. Guy, G. J. Popek, J.-F. Paris, and H. Garcia-Molina. Primarily disconnected operation: Experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, November 1992.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[11] P. J. Keleher and U. Cetintemel. Consistency management in Deno. *Mobile Networks and Applications*, 5(4):299–309, 2000.

[12] M. Kim and B. D. Noble. Mobile network estimation. In *7th ACM Conference on Mobile Computing and Networking*, July 2001. to appear.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[14] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[16] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Technical Conference*, pages 95–106, New Orleans, LA, January 1995.

[17] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX 1997 Annual Technical Conference*, pages 275–288, Anaheim, CA, January 1997.

[18] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, CO, December 1995.

[19] B. Noble, B. Fleis, and M. Kim. A case for Fluid Replication. In *Network Storage Symposium*, Seattle, WA, October 1999.

[20] B. D. Noble, B. Fleis, and L. P. Cox. Deferring trust in Fluid Replication. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[21] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–149, Nashville, TN, May 1994.

[22] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM 97 Conference*, pages 51–61, Cannes, France, September 1997.

[23] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software — Practice and Experience*, 28(2):155–180, February 1998.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *15th ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[25] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137–152, Boston, MA, June 1994.

[26] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, Saint Malo, France, October 1997.

[27] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[28] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994. Corrigendum: 12(2):165–72, 1994.

[29] D. C. Steere, J. J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the Sun vnode interface. In *Proceedings of the 1990 Summer USENIX Conference*, pages 325–331, Anaheim, CA, 1990.

[30] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, September 1994.

[31] R. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the IEEE 4th Workshop on Workstation Operating Systems*, pages 71–78, Napa, CA, October 1993.

[32] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

[33] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 75–84, San Diego, CA, October 2000.

Obtaining High Performance for Storage Outsourcing

Wee Teck Ng, Bruce Hillyer, Elizabeth Shriver, Eran Gabber, Banu Özden
Information Sciences Research Center
Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
{weeteck, bruce, shriver, eran, ozden}@research.bell-labs.com

Abstract

Storage outsourcing is an emerging industry that shields storage users from the complexity of in-house storage management, while providing cost savings and reliability improvements via the aggregation of storage into large, special-purpose facilities. These distributed and replicated facilities are operated by a storage service provider, and are accessed by remote users via high-speed network connections.

The viability of storage outsourcing is critically dependent on the performance of remote storage. In this paper, we measure the performance of I/O benchmarks accessing a remote block-level storage system. We use benchmarks that represent a variety of workloads, running on several operating systems and file systems. Network latencies represent distances ranging from a local neighborhood to halfway across a continent. We vary the network loss characteristics to correspond with the conditions of either dedicated fiber or shared Internet (with loss rates up to 10^{-3}). We examine the effectiveness of basic latency-hiding techniques such as caching, application prefetching, and asynchronous writes. We conclude that remote storage is already viable for a wide variety of active workloads, and we point out areas where new techniques could provide significant additional performance enhancement.

1 Introduction

Storage management is complex and expensive. For example, IDC estimates that for every dollar spent on storage equipment, an additional $6 will be spent on managing the storage [16]. This includes the expert help required to configure the storage systems (e.g., host, RAID, and SAN configuration, cabling, and cooling), to administer it (backup and restore), and to manage it for high availability (capacity planning, disaster recovery). These problems are motivating the emergence of storage service providers, who sell data storage as an outsourced business service. Among the major storage service providers are traditional computing system suppliers (e.g., IBM, HP), telecommunication vendors (e.g., Qwest), and startups such as StorageNetworks Inc. (www.storagenetworks.com).

Remote storage has a long history for applications such as distributed databases and FTP archives, but dramatic improvements in the price and availability of high-speed networking suggest that a much broader scope of applications may thrive in an environment of outsourced storage and commercial storage service providers. The viability of the emerging storage outsourcing industry depends, in part, on whether acceptable performance can be obtained from remotely-accessed storage, but the open literature lacks technical data on the performance of remote storage systems. Can remote storage substitute for host-attached disks, nearby storage area networks, or LAN-based network-attached storage servers?

To explore this question, we measure a variety of benchmarks widely used in the file system and database communities. Our experimental platform consists of PCs, a fiber-based gigabit Ethernet, a router testbed, and our own SCSI over IP implementation, which predates standards, but is largely comparable to iSCSI. We measure benchmarks on an accepted kernel tool (the FreeBSD dummynet package) that introduces "network delay" and loss into the protocol stack. On this platform we measure benchmarks when network propagation delay ranges up to 8 ms, corresponding to 1600 km of fiber. We also investigate the performance of outsourced storage under the network delays and packet losses characteristic of the Internet, using a testbed that consists of two Cisco routers with an OC-3 backbone, and a pair of Smartbits generators that generate background traffic

consistent with a traffic profile derived from recent Internet traffic studies [4, 30, 21].

We observe that in lossless network conditions, the remote storage behaves in many respects like a local disk that has a moderately slow access time: the traditional caching, application prefetching, and asynchronous write techniques are typically effective in hiding the access delays. For high performance under loss and delay characteristics similar to the Internet, I/O-intensive benchmarks may require a large cache, network protocol tuning, and network support for packet prioritization.

The structure of this paper is as follows. Section 2 covers related work, Section 3 describes our remote storage testbed and benchmarks, Section 4 presents the performance measurements and discusses techniques to overcome network latency and congestion in a remote-storage environment, and Section 5 gives concluding remarks.

2 Related Work

2.1 Storage over IP Protocols

A key component of storage outsourcing is the transport of stored data over a communication network. Current storage service providers offer raw data block service over local storage-area networks (SAN) and over wide area networks (WAN). They typically use proprietary products such as EMC's SRDF software to replicate data to remote storage [6], and rely on a media-specific protocol (e.g., Fibre Channel protocol, ESCON, ATM) as the transport protocol. An emerging alternative for remote storage access is to encapsulate SCSI disk commands and data in IP packets. This approach is the target of a standardization effort by the Internet Engineering Task Force. iSCSI [24] is a SCSI over TCP/IP protocol proposed by a group including IBM, Cisco, HP, Quantum, SanGate and 3Com. It enables clients to address SCSI devices directly over an IP network. The protocol provides flow control, a method to include phase and tag information in a TCP stream, target buffer management, and resource discovery and management. Several working prototypes have been demonstrated in the InterOperability Lab at the University of New Hampshire [11]. In October 2001, the current iSCSI draft defines the encapsulation mechanism, message format, and session management, but many additional aspects are still under discussion.

Prior research projects have studied the use of IP over LAN or SAN for storage applications. The NASD project at CMU developed a storage architecture that enables direct client access to storage on a LAN [9]. The storage device exports an object interface, and allows clients to read and write to it directly, after securing the proper security credentials from an object manager. Prototypes implemented on Alpha workstations using RPC over UDP/IP demonstrated performance comparable to server-attached disks, but with better scalability. The Netstation project at USC studied the feasibility of using IP as a transport protocol between host and peripherals (e.g., storage devices) [32]. They implemented prototypes on Sun workstations with UDP/IP, and showed that it is possible to achieve 80% of SCSI's maximum throughput without the use of network coprocessors.

2.2 File System Research

Remote storage raises many of the traditional problems seen in file system research. Many file system techniques such as caching and prefetching [15] are effective in hiding the delays associated with access to local disks, so we expect them to be similarly helpful to the performance of remote storage. More recent file system research on minimizing the number of file system synchronous writes [8] and reducing write latency for small writes [33] will also alleviate the impact of network delay on data written to remote storage. See [25] for a summary of recent work in this area.

Most distributed file systems employ latency-hiding techniques to mitigate the effects of network delay on client performance. These techniques include caching data at the client, reducing the number of synchronous writes, and reducing the protocol overheads. For example, NFS v3 [20] uses asynchronous writes with commit. NFS v4 [26] groups related commands into a single compound command to reduce the number of round-trip times. It also delegates data ownership to clients to enable more aggressive client-side caching. Martin et al. [14] analyze the effect of network delay on the SPECsfs benchmark running on NFS v3. They verify their measurements analytically with a queuing model. Their results indicate that NFS is

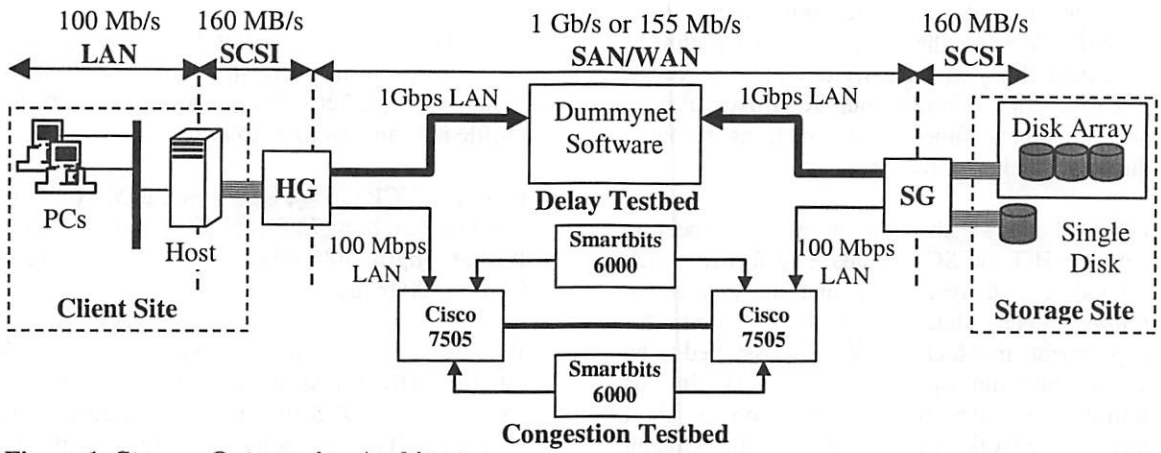


Figure 1: Storage Outsourcing Architecture.

insensitive to network latency up to 150 µs, and that performance decreases linearly with increasing delay.

A large body of research and commercial products show how to implement file systems on top of shared block storage [29]. These works complement our research and can provide the front end to our storage system.

Our paper extends previous work in several ways. We give measurements of SCSI over IP access to remote storage for network latencies corresponding to distances up to 1600 km, for a variety of I/O-intensive low-level and application-level benchmarks. In this context, we compare the performance on several operating systems and file systems, we examine the consequences of packet loss and congestion as seen in the Internet, and we explore the impact of techniques such as server-side and client-side caching and suppression of synchronous writes.

3 Remote-Storage Architecture

In our lab we have several projects that investigate broad issues in remote storage, and that deal with a variety of topics such as storage virtualization, remote replication and failure recovery, the partitioning of storage resources to serve concurrent clients, and implementing all the details of the emerging iSCSI protocol. For the performance experiments described in this paper, we use a simplified remote storage architecture that that consists of a single client site that is networked to a single storage site, as depicted in Figure 1.

3.1 System Description

The center portion of Figure 1 shows a network that connects a pair of machines called the host gateway (HG) and the storage gateway (SG). These machines implement SCSI over IP, and in experiments in Section 4.4 they also perform caching. (In a full storage system, the HG and SG machines would also implement mechanisms such as virtualization, replication, and recovery. In small systems, the HG and SG functionality might be implemented by cards rather than by separate PCs.) On the left side of the figure, the host connects to the HG via a standard SCSI cable, on which the HG appears to be one or more local SCSI disks. On the right side of the figure, the SG behaves like a local host as it accesses standard SCSI disks or RAID arrays on behalf of the HG.

The host, the HG, and the SG are PCs equipped with an Intel 440GX motherboard, dual 700 MHz PIII CPUs, 768 MB SDRAM, Intel 100 Mb/s Fast Ethernet, and Adaptec 29160 Ultra160 SCSI adapter cards. All PCs run FreeBSD OS 3.4 (except for experiments we describe that measure other operating systems). We run most benchmarks directly on the host, but the Surge and TPC-C benchmarks use a 3-level architecture in which a pool of clients access the host machine as a server, which in turn accesses remote back-end storage via the HG. The Surge clients access the host server via http. The TPC-C clients access the host database server via SQLNet. For Surge and TPC-C, the clients are PCs equipped with Intel 440BX motherboard, 400 MHz PII CPU, 64MB SDRAM, and Intel 100 Mb/s Fast Ethernet.

Our version of SCSI over IP for these experiments predates the IETF iSCSI draft [24]. We wrap SCSI messages according to the iSCSI

message format, and transport them over TCP/IP. We use the immediate data option to send data along with the **write** command where possible. This version of our SCSI over IP does not implement functionality such as resource discovery and error recovery.

Figure 1 shows that the network connection between HG and SG can take two forms, which we call the *delay testbed*, and the *congestion testbed*. The delay testbed is used for experiments in which the WAN is assumed to be a dedicated high-speed fiber network. In this setting, we take measurements on a short physical network (a Lucent P550 Cajun Gigabit Ethernet Switch between a pair of Alteon ACE Gigabit Ethernet cards) that is augmented with the FreeBSD dummynet tool [22]. Dummynet is a kernel tool that modifies the behavior of the network protocol stack in ways that accurately mimic network behavior with respect to queuing, bandwidth limitations, delays, and packet losses. We use dummynet to vary the one-way propagation delay from 0 ms to 8 ms to reflect the performance impact of network distances up to 1600 kilometers. We calibrate the dummynet settings by measuring the performance of the `ping` command over a pair of actual fibers that form a 60 km bidirectional link (0.3 ms propagation delay each way). Although we use Gigabit Ethernet as the storage interconnect in the delay testbed, we believe that our findings would be similar over alternative storage interconnects such as Fibre Channel.

The congestion testbed simulates the congestion and packet loss of the Internet. The backbone consists of a pair of Cisco 7505 routers connected via an OC-3 link (155 Mb/s). The edge access networks are Fast Ethernets (100 Mb/s). We use a pair of Smartbits hardware traffic generators to impose background traffic on the backbone. Smartbits is more controllable and accurate than is a host-based generator such as *tcplib* [5] — a host-based TCP generator will back off under high load, thus failing to maintain the desired background load, whereas Smartbits can be programmed to maintain a fixed load distribution. Similarly, it would be difficult to simulate network congestion accurately via a simple tool like dummynet, because the various network parameters (e.g., delay and error rate) vary in a non-linear way with increasing load.

Our traffic profile represents current Internet behavior, by contrast with earlier networking studies based on telnet and ftp traffic (e.g., [2]). We obtain detailed statistics for bytes, packets, and flows from [21, 4], and second-order statistics on packet delay from [30]. Our traffic profile is summarized as follow:

Protocol: TCP (95%), UDP (5%), ICMP (<1%).
Application: http (75%), ftp (5%), misc (20%).
Packet length: 50% <44 bytes, 75% <576 bytes, 99% <1500 bytes.

We measure three types of storage devices: an 8 GB IBM DNES disk with 2 MB disk cache, an 18 GB IBM DDYS disk with 4 MB disk cache [10], and a Terrasolution disk array [3] with 128 MB disk cache and eight 18 GB IBM DDYS disks configured as RAID 5 with a 16 KB chunk size. We enable write cache on the disk, assuming that recovery is handled elsewhere (e.g., using NVRAM in the HG.)

Our measurements are conducted on 3 operating systems and 4 file systems. The bulk of our experiments are run on FreeBSD OS v3.4, which has a tunable file cache and a separate VM cache that holds clean data only. On this platform we measure the traditional UNIX FFS that uses synchronous metadata updates to ensure that it can recover the file system data structures to a consistent state after a crash, and we measure the more modern Soft Updates FFS, which uses careful update ordering to reduce the number of synchronous writes while still ensuring integrity [25]. We also repeat selected benchmarks on Microsoft Windows NT4.0, which has a small (and non-tunable) file cache, and on Windows 2000, which has an integrated file and VM cache. The Windows measurements are conducted on both the FAT file system, which uses synchronous writes to maintain metadata integrity, and on the more modern NTFS, which is a journaling file system that avoids synchronous writes by logging metadata updates to two log files on the disk [28].

3.2 Benchmarks

We choose our benchmarks to represent a wide variety of workloads that could access remote storage. The goal of our evaluation is to understand the impact of network delays, file system features, and storage system parameters on application performance when using remote storage. Each data point is obtained from 10 experimental runs. About 2 months are required

to run 10 iterations of the full suite of experiments.

We begin with microbenchmarks that measure basic read and write performance, for sequential, random, and identical-block access patterns. (The latter access pattern reveals the time for a hit in the disk's cache.) These microbenchmarks access the raw disk device (i.e., the remote storage that is made to look like a local raw disk by our SCSI over IP software). We use a single-threaded program that reads or writes blocks that range from 1 KB to 64 KB. Our working set is 4 times larger than the remote storage system's cache. The performance metric is bandwidth (**Mbytes/sec**). We are able to model these simple I/Os analytically, and we use these models to give insight into the performance of the application benchmarks described below.

The SSH benchmark [25] represents a software development workload. It unpacks, configures, and builds a software package named SSH. The unpack phase extracts a compressed tar archive containing the source tree (383 files, totaling about 65,000 lines of commented code). It thus reads a large file sequentially and generates many subsequent small metadata operations. The config phase determines what features are available on the host operating system and generates a makefile. To do this, it compiles and executes many small test programs. Because most of the operations are on small files, we expect many metadata operations. The build phase executes the makefile to build the ssh executable. We run the three phases of the benchmark consecutively, so the config and build phases run with the file system cache warmed by the previous phases. We use throughput as a performance metric, by converting the time to run a single SSH benchmark iteration into **iterations/hour**.

The SDET benchmark [7] is designed by SPEC to simulate a typical timesharing workload. It models a software development environment (e.g., editing, compilation, and various UNIX utilities), and makes extensive use of the file system. SDET runs scripts that execute a predetermined mix of commands, and the reported metric is **scripts/hour**, as a function of the number of scripts running concurrently. We report results with 32 concurrent scripts.

The Surge benchmark [1] uses a workload generator to simulate a set of users accessing a web server. It is parameterized by empirical statistics extracted from web server logs. The parameters model server file size distribution, request size distribution, relative file popularity, embedded file references, temporal locality of reference, and idle periods of individual users. We run the benchmark with default settings, except that we increase the data size and load by using 20,000 files, and 4 clients, each running 2 processes and 50 threads per process. These parameters represent a workload with 400 timesharing users accessing a 1 GB data set. The performance metric is **operations/sec**.

The PostMark v1.1 benchmark [12] simulates the workload seen by Internet Service Providers under heavy load: a combination of electronic mail, netnews, and web-based e-commerce transactions. It creates a large set of files with random sizes within a set range, and then executes a large number of file create, delete, read and append operations. We set the benchmark to run with 250,000 files and the default size range (512 bytes to 16 KB), giving a working set of 500 MB. The performance metric is **transactions/sec**.

The TPC-C benchmark [31] simulates an online transaction-processing database workload. It models a wholesale supplier managing orders, and a workload consisting of a specified mix of five transaction types. We store the database in an Oracle 8i database v8.1.7 Enterprise Edition, and run experiments on Windows NT Server 4.0 SP6, and on Windows 2000 Server 5.0. The database is created using default settings recommended by Oracle. The client program is written in the PL/SQL language, adapted from the TPC-C sample program in Appendix A of [31]. We use a scaling factor of 30 to size the database, resulting in a working set of 2 GB (including log files). The database is restored to the same state prior to each run using a disk-level restore command. The complete database (including initialization files) resides in a single file system on the remote storage. For convenience, and as recommended by Oracle, we store the database in a file system rather than in a raw disk partition. Accessing the database through raw I/O instead of through a file system could improve performance by 5–10% [19], but this would not affect our conclusions. The performance metric is **transactions/min**.

4 Performance Results

4.1 Microbenchmark Results

To see the basic effects of network delays, and to discover where the system overheads are, we begin with measurements of blocking 8 KB reads and writes from a single thread that runs on the host PC. We measure I/Os to a local SCSI disk on the host, (reported in the row labeled "local" in Table 1), and I/Os to a local SCSI disk on the HG machine (the row labeled "target"), and I/Os to storage on the SG under various network delays. We conduct measurements on several kinds of storage devices: an IBM DNES disk, an IBM DDYS disk, a RAID 5 array, and /dev/zero (the latter enables us to separate out storage hardware delays from SCSI over IP processing). We also use in-kernel timing measurements to approximate the latency of each component between the host and the remote disk. These latter measurements are given in Figure 2.

The basic performance can be modeled simply, as described in more detail in [18], via this equation.

$$T_{IOLatency} = T_{disk} + 2 \times T_{netDelay} + T_{iSCSI} \quad (1)$$

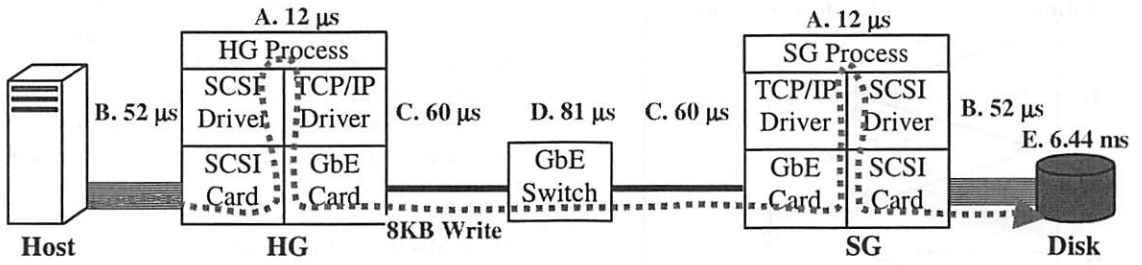
For random I/O, T_{disk} is the disk access time (seek, rotation and command processing) plus the data transfer time (i.e., number of bytes divided by media transfer rate) [10]. For sequential I/O, T_{disk} omits the seek and rotational components. $T_{netDelay}$ is the **one-way** network latency, and T_{iSCSI} is our SCSI over IP protocol processing latency. From Figure 2, we see that, in the case of an 8 KB data transfer to the disk with a 64 B acknowledgement from the SG, T_{iSCSI} is 0.408 ms.

In Table 1 we can see the round-trip overhead of our SCSI over IP implementation in the response time section — it is the difference between the local row and the 0 ms row. The overhead generally ranges from 0.4 to 0.6 ms, which is in

Table 1: Microbenchmark performance (8 KB record size) on FreeBSD. The working set is 4 times the disk cache size for disk-based storage systems, and 1 GB for the null device. Performance is expressed in throughput and response time, and we report the average of ten runs. The standard deviation is less than 2% of the mean; the maximum deviation is 6%. **Local** disk is a server-attached disk, **Target** disk resides on the HG and does not incur SCSI over IP overheads. We vary the **one-way** network delay from 0 to 8 ms.

Throughput (MB/s) for Microbenchmark with 8KB record Size																
Net. Delay (ms)	1 Disk, IBM DNES				1 Disk, IBM DDYS				Disk Array, 8xDDYS				Null Device			
	Random		Sequential		Random		Sequential		Random		Sequential		Random		Sequential	
	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Local	0.77	0.70	13.94	20.00	1.07	1.57	34.04	29.40	1.01	3.38	19.52	16.94	1224.18	2501.56	1436.52	3555.56
Target	0.76	0.70	13.26	19.19	1.05	1.40	28.97	29.86	1.00	3.14	12.83	12.94	49.02	51.22	52.51	51.85
0	0.73	0.70	9.86	9.75	0.97	1.43	10.40	9.93	0.95	2.92	8.48	8.43	18.16	18.71	17.96	19.17
1	0.62	0.70	2.84	2.89	0.79	1.40	3.00	2.91	0.77	2.67	2.48	2.77	3.25	3.26	3.28	3.29
2	0.53	0.70	1.65	1.67	0.66	1.41	1.70	1.67	0.62	1.59	1.58	1.63	1.79	1.77	1.80	1.82
4	0.42	0.67	0.90	0.91	0.49	0.90	0.91	0.90	0.47	0.88	0.88	0.88	0.95	0.95	0.94	0.95
8	0.29	0.46	0.47	0.47	0.33	0.47	0.47	0.47	0.32	0.47	0.47	0.47	0.47	0.46	0.46	0.46

Response Time (ms) for a single 8KB I/O																
Net. Delay (ms)	1 Disk, IBM DNES				1 Disk, IBM DDYS				Disk Array, 8xDDYS				Null Device			
	Random		Sequential		Random		Sequential		Random		Sequential		Random		Sequential	
	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Local	10.38	11.44	0.57	0.40	7.50	5.10	0.24	0.27	7.89	2.37	0.41	0.47	0.01	0.00	0.01	0.00
Target	10.50	11.45	0.60	0.42	7.60	5.70	0.28	0.27	8.02	2.55	0.62	0.62	0.16	0.16	0.15	0.15
0	10.99	11.38	0.81	0.82	8.24	5.61	0.77	0.81	8.43	2.74	0.94	0.95	0.44	0.43	0.45	0.42
1	12.94	11.48	2.82	2.77	10.09	5.71	2.67	2.75	10.35	3.00	3.22	2.89	2.46	2.46	2.44	2.44
2	15.11	11.44	4.86	4.80	12.21	5.67	4.71	4.80	12.82	5.02	5.08	4.92	4.48	4.52	4.46	4.40
4	19.04	11.89	8.92	8.82	16.29	8.93	8.79	8.86	17.13	9.06	9.08	9.09	8.43	8.42	8.50	8.41
8	27.18	17.47	17.10	17.04	24.46	17.09	16.99	17.02	25.32	17.20	17.17	17.20	17.20	17.35	17.35	17.35



$T_{iSCSI} = 2\times(52\mu s + 12\mu s + 60\mu s) + 81\mu s + 2\times(1\mu s + 10\mu s + 15\mu s) + 27\mu s = 408\mu s$

	Module	Data Size		
		64 B	1.5KB	8KB
A	HG/SG Process	10 µs	12 µs	12 µs
B	SCSI (CAM+Adaptec 2960)	1 µs	11 µs	52 µs
C	IP (TCP/IP_Alteon NIC)	15 µs	21 µs	60 µs
D	GbE Switch (Lucent Cajun)	27 µs	50 µs	81 µs
E	Disk Array (Hit in Cache; Random Blocks)	0.27; 5.65 ms	0.29; 5.75 ms	0.38; 6.44 ms
E	IBM DDYS Disk (Hit in Cache; Random Blocks)	0.20; 4.67 ms	0.21; 4.75 ms	0.29; 5.09 ms

Figure 2: Data flow and Latency estimates for a single 8KB I/O in our SCSI over IP prototype.

approximate agreement with the sum of the component overheads shown in Figure 2 (about 0.4 ms). In the throughput table, the null device measurements with 0 ms network latency show that the overhead of this SCSI over IP implementation limits the throughput of a single blocking thread to about 18 MB/s for 8 KB I/Os. The corresponding figure for 64 KB I/Os is about 30 MB/s.

The throughput table shows a huge drop in sequential I/O performance from network delay. The reason is that the sequential workload hits in the disk drive cache, so any network delay gives a huge percentage increase in total service time. For instance, consider sequential writes to the single DDYS disk. The throughput is 34.04 MB/s for the local disk, 10.4 MB/s for a 0 ms propagation delay, and 3.00 MB/s for a 1 ms propagation delay. This performance drop is explained by the response time table. At 34 MB/s, the I/O response time is 0.24 ms (the disk's cache acks the write immediately). Adding an overhead of 0.5 ms (for SCSI over IP) increases the total response time to 0.77 ms: SCSI over IP increases the delay by a factor of 3, which cuts the throughput by a factor of 3. Adding an additional 1 ms of propagation delay each way increases the total latency to 2.75 ms: When the latency grows from 0.24 ms to 2.75 ms, the throughput drops proportionally, from 34.04 MB/s to 3 MB/s. In general, for the sequential I/O microbenchmark the response time doubles as the network latency doubles. The random workloads are less severely impacted by network latency, because the disk random-access latency dilutes the impact of the network latency.

For single disk configurations (shaded columns in Table 1), random write performance is relatively insensitive to network delay. Figure 3 explains this behavior by analyzing the event sequence that occurs when writing to a busy disk. At t_1, a write request reaches the disk and the disk generates a SCSI status message immediately. The disk starts to flush the data to media since its cache is full. At t_2, another write reaches the disk, but it must wait until the previous write completes at t_3, freeing up space in the cache. Thus, if the roundtrip delay ($2\times T_{netDelay}$) is less than the disk response time (T_{disk}), the network delay is masked by the disk latency and the client's response time remains constant (i.e., $T_{disk} + T_{iSCSI}$). If the roundtrip delay is greater than the disk latency, the client's response time is only dependent on the network delay (i.e., $2\times T_{netDelay} + T_{iSCSI}$). This behavior is not seen in the disk array because of its larger cache and much higher aggregate write bandwidth.

For an I/O size smaller than the disk array chunk size (16 KB), the disk array overhead generally impairs the performance of a single-threaded request stream. In the case of a workload that has concurrent I/O requests, we would expect the aggregate performance of the array to be better than that of a single disk.

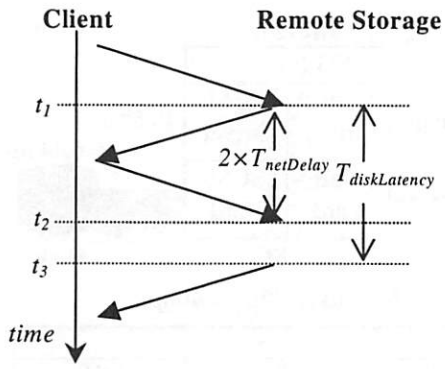


Figure 3: Writing a Single Block to a Random Disk Address.

4.2 Application-Level Benchmarks

The microbenchmark measurements in the previous section indicate that network latency impairs the performance of a single-threaded process that spends 100% of its time trying to do I/O from remote storage. We next consider the performance of a variety of application-level I/O benchmarks, to see the performance of remote-storage on more typical workloads. Except where stated otherwise, the measurements in this section are obtained on a remote storage system that uses a disk array (not a single disk).

4.2.1 Experiments that vary network latency

Table 2 shows the application benchmark performance versus network delays, for the FreeBSD OS with the traditional FFS and the Soft Updates FFS file systems, and for Windows NT and Windows 2000 with the NTFS and the FAT file systems. We measure I/Os to storage on the host, (the row labeled "local"), I/Os to storage on the HG machine (the row labeled "target"), and I/Os to storage on the SG under **one-way** network propagation delays ranging from 0 to 8 ms.

We make the following observations:

1. **Our SCSI over IP implementation overhead is small.**

We observe that our SCSI over IP implementation adds little overhead, both in terms of application performance (Table 2), and host CPU usage. The performance degradation from a host-attached disk to a target disk averages 6% across all the benchmarks (comparing rows labeled **Local** and **Target**). We experienced higher overheads (18%) when accessing a remote disk over the IP protocol stack (comparing rows labeled **Local** and **0 ms**). The CPU usage at the HG is less than 10% for all workloads, implying that we do not need a high performance processor for the SCSI over IP gateway—it would be feasible to implement the gateway on a network interface card.

2. **Network latency has little effect on web, database, and CPU-bound benchmarks.**

The Surge benchmark models a typical web server, where hot documents are accessed very frequently, and thus many accesses hit in the web server's local cache. Consequently, the workload seen by the remote storage consists largely of asynchronous sequential writes to log files. Thus, we observe very little sensitivity to network latency. For the TPC-C benchmark, we observe

Table 2: Application benchmark performance. The **local** row gives the throughput (as defined for each benchmark in Section 3.2) of a host-attached disk array. Other rows give the normalized throughput as a fraction of the local row. The **target** row accesses an array on the HG, without SCSI over IP overheads. Subsequent rows vary the one-way network delay from 0 to 8 ms. **Soft** is the Soft Updates FFS file system. We report the mean over ten runs: the standard deviation is less than 5% of the mean; the maximum deviation is 16%.

Net. Delay (ms)	FreeBSD 3.4								Windows 2000				Windows NT Server 4.0			
	SSH		SDET		PostMark		Surge		PostMark		TPC-C		PostMark		TPC-C	
	Soft	FFS	Soft	FFS	Soft	FFS	Soft	FFS	FAT	NTFS	FAT	NTFS	FAT	NTFS	FAT	NTFS
Local	67.3	60.0	4253.7	3376.9	386.2	361.9	3058.0	2765.9	326.3	237.6	730.7	727.1	80.0	61.7	750.1	733.5
Target	0.94	0.96	0.94	0.84	0.91	0.95	0.96	0.96	0.96	0.89	1.01	1.02	0.98	0.91	0.95	0.97
0	0.89	0.98	0.70	0.49	0.56	0.57	0.97	0.84	0.70	0.55	1.00	1.02	0.94	0.92	0.94	0.95
1	0.87	0.90	0.33	0.15	0.20	0.17	0.92	0.84	0.42	0.31	0.99	1.04	0.74	0.79	0.93	0.94
2	0.83	0.74	0.18	0.08	0.11	0.14	0.90	0.92	0.28	0.25	0.98	0.97	0.50	0.54	0.93	0.93
4	0.78	0.61	0.10	0.04	0.06	0.07	0.89	0.92	0.14	0.13	0.93	0.93	0.32	0.35	0.88	0.90
8	0.69	0.49	0.05	0.02	0.03	0.03	0.86	0.88	0.07	0.07	0.88	0.89	0.17	0.19	0.82	0.84

that the performance is immune to small network delays (<2 ms). Performance only degrades modestly with larger network delays, because a modern DBMS is designed to overcome I/O latency. It uses a high multiprogramming level to generate numerous concurrent I/Os, and clever caching and prefetching to reduce the number of I/Os that block progress [27]. The database may prefetch tables to gain the benefit of large sequential reads, and may use vertical partitioning of tables to access and cache only the required columns. Consequently, the database's performance will not suffer, given sufficient concurrency and I/O bandwidth. Although the SSH benchmark has been proposed as an I/O benchmark [25], we find that SSH is mostly CPU bound, especially during the configure and build phases. Thus, SSH's performance decreases by only 33% as the one-way propagation delay increases from 0 ms to 8 ms, versus 10× for PostMark and SDET.

3. **Read traffic is less significant than write traffic.**

Many of these benchmarks (SSH, SDET, PostMark) have little read traffic that reaches the disk, as they first generate the data needed in subsequent phases of the benchmarks, and much of their working sets fit in main memory. Thus most traffic seen at the disk level is writes. (Many file system workloads have this characteristic because of the effectiveness of read caching.) TPC-C has significant read traffic (35–43% of overall bytes transferred). But most of these reads are either prefetch or not in the critical path. The DBMS prefetches whole tables during the early portion of the benchmark. These prefetches are mostly asynchronous reads from sequential locations, so they are serviced quickly. Because the TPC-C working set is larger than the buffer cache size, pages are subsequently evicted from the database buffer cache during the course of the run. Nevertheless, the number of synchronous reads gradually decreases during the benchmark run. The write traffic in TPC-C appears to be mostly delayed writes and not in the critical path, and thus has little impact on the application performance.

4. **Some applications exhibit super-linear degradation (> 2x when delay doubles).**

With the FFS file system, PostMark and SDET exhibit a large drop in performance from the local array to remote storage with 0 ms and 1 ms propagation delays (see shaded cells in Table 2). This is the same phenomenon as seen in the large throughput drops for sequential I/O in the microbenchmarks. For PostMark and SDET, the predominant I/O is single-threaded random writes to metadata. These writes take 0.4 ms in the disk array (they are ack'd from the array cache). Since our SCSI over IP overhead is about 0.4 ms for 8KB writes, the degradation is 50% from a local array to a remote array with 0 ms propagation delay. When the propagation delay increases to 1 ms (2 ms round-trip), the response time for a single 8KB I/O increases to 2.8ms, and the performance drops accordingly.

4.2.2 Experiments that vary network congestion

In this section we present the results of the PostMark and TPC-C benchmarks when measured on the congestion testbed with the Smartbits traffic generator applying a background network load. PostMark is I/O bound, and the insights that we obtain from it are

Table 3: Effects of Congestion on Application Performance.

Load Factor (%)	Background Traffic Profile				PostMark (Trans/sec)			TPC-C (trans/min)
	BW	σ(BW)	Delay	PktLoss	BSD FFS	BSD SoftUpd	NT4.0 NTFS	NT4.0 NTFS
0	0	0	0.46	0	64	80	52	722
20	31	9.8	0.54	0	63	79	49	719
30	40	11.7	0.54	2×10^{-6}	60	75	46	720
35	43	12.3	0.55	5×10^{-6}	59	74	48	718
40	56	12.7	0.64	2×10^{-5}	56	73	47	719
50	76	12.8	0.70	5×10^{-5}	53	72	47	717
55	86	12.8	0.72	7×10^{-5}	51	63	46	710
60	96	12.8	0.78	1×10^{-4}	44	54	42	700
70	106	12.8	1.07	3×10^{-4}	26	29	26	693
75	110	12.8	1.68	7×10^{-4}	No Data			668

applicable to other I/O-bound applications. The TPC-C results demonstrate the effectiveness of application-level latency hiding techniques.

Table 3 shows the results of PostMark on FreeBSD with FFS and Soft Updates, and on Windows NT Server 4.0 with NTFS. This table also shows TPC-C results on NT Server 4.0. We measure the performance of these benchmarks as the background network traffic ranges from 0 to 75% of the backbone link capacity. The background traffic is based on the traffic profile described in Section 3.1, and is characterized by 4 parameters: **BW** is average bandwidth in Mb/s, **σ(BW)** is standard deviation of the measured bandwidth, **Delay** is round trip propagation delay in ms, and **PktLoss** is average packet loss rate. The shaded region in the table indicates network congestion (i.e., the application is starved for network bandwidth).

With no background load, the benchmark performance is limited by the fixed 0.46 ms equipment delay inherent in the router network. As the background load increases, we observe that the application's performance decreases gradually until the onset of congestion. The bandwidth requirement of the application varies from 56 Mb/s for PostMark under FreeBSD OS to 24 Mb/s for TPC-C. When the background traffic grows so large that it begins to take bandwidth that is required by the application (shaded region in Table 3), the application becomes *both* delay and bandwidth bound, so performance deteriorates significantly. Over the Internet, we may need more than caches and synchronous write suppression to obtain good remote I/O performance.

4.3 Identifying bottlenecks in a SCSI over IP system

In this section we analyze various components in our SCSI over IP system to identify areas for improvement. This includes the host, the edge access equipment (HG and SG), the network, and the remote storage system.

4.3.1 Host

The host has several components affecting performance, including the file system, OS, network protocol stack, and SCSI protocol stack.

File system: We observe from Table 2 that the file system design affects performance. The results for Soft Updates versus traditional FFS show the benefit of suppressing synchronous I/Os. For instance, PostMark runs 22–33% faster with Soft Updates under all network delay conditions.

OS: We focus on host buffer cache design as it determines the amount of traffic seen at the disk level. We find that a small (and static-size) file cache is bad. This can be seen by comparing the performance of PostMark on NT Server 4.0 and on Windows 2000. NTFS is supposed to have an integrated VM cache that is dynamically resized according to workload, and its cache size can be controlled by the user (e.g., by setting the *LargeSystemCache* parameter [28] or by using the public domain program *CacheSet* [23]). However, we estimate using the Windows Task Manager tool that the file cache size in NT never goes beyond 10% of the working set under various I/O loads and *CacheSet* settings. The resulting cache capacity misses significantly degrade performance, as seen in PostMark, which runs 4 times slower on NT than on Windows 2000. An application can compensate for a small host cache by caching data in application buffers, as is done in the Oracle database: the TPC-C performance numbers on the NT and Windows 2000 platforms are comparable despite the small host buffer cache in NT. We also find that the number of file cache entries should be dynamically tunable. FreeBSD has a small buffer cache (default 512 buffer cache entries) that is fixed at kernel build time. It also has a large VM cache that holds only clean pages. This limited amount of buffer available to cache asynchronous write traffic causes I/O-bound applications running on Soft Updates to stall during high network delays while waiting for buffer cache cleaning. This is seen in PostMark: the performance of traditional FFS converges with that of FFS with Soft Updates as the delay increases (see Table 2), even though FFS with Soft Updates has very few synchronous I/Os.

TCP/IP: TCP is responsible for 37% of the latency for an 8 KB transfer to remote storage when the network propagation delay is 0. This assumes that we have large TCP windows to minimize unnecessary fragmentation of SCSI data, thereby avoiding extra round-trip delays. Most researchers consider window size to be a critical bottleneck for networks having a high

bandwidth-delay product [13]. We find that most application-level benchmarks use record sizes of 4–8 KB (metadata). Larger I/Os (16–256 KB) are mainly asynchronous data transfers and are not in the critical path. In our experiments we verified that a default window size of 64 KB does not harm performance by comparison with a larger window size of 640 KB. However, window size may be important if the application or file system generates a significant volume of synchronous I/O traffic. A related issue is the packet size in the network layer (i.e., the MTU size). To reduce fragmentation overhead, we prefer to use the largest MTU allowable by the network. For example, increasing the MTU from 1.5 KB to 8 KB (i.e., jumbo frame) gives a 15% performance improvement in the 8 KB random-write microbenchmark. We also can verify this from Figure 2, by calculating the 8 KB I/O latency (0.461 ms versus 0.408 ms). In application level benchmarks, using a 1.5 KB MTU instead of 8 KB degrades the performance by 1%–9% when the propagation delay is low (i.e., SAN environment); the degradation is less severe with higher propagation delays (>1 ms). Lastly, we find that data copying in the host protocol stack (<10 µs for 8 KB) is insignificant as it is much smaller than other latencies (e.g., 81 µs for 8 KB in the GbE switch).

SCSI: SCSI over IP is responsible for 25% of the latency for an 8 KB transfer to remote storage when the network propagation delay is 0. This assumes that we use the immediate data option that avoids an extra round trip delay that would be incurred if the command were sent separately from the data [24].

Our results indicate that hardware assistance for SCSI and TCP/IP protocol processing may be beneficial in a SAN. In the SAN, the network propagation delay is on the order of 10 µs (2 KM of fiber), whereas our measured protocol processing time for an 8 IB I/O is 53 µs for SCSI, plus 75 µs for TCP/IP. By contrast, in a WAN environment the protocol processing overhead is insignificant: the propagation delay dominates.

4.3.2 Network

Our edge access equipment (HG and SG) is responsible for 327 µs of latency per 8 KB I/O (see Figure 2). Thus, in a SAN environment, replacing the HG and SG by a native implementation of iSCSI in a host adapter and in the remote storage array could significantly improve the performance of remote storage. The network equipment latency (i.e., 81 µs per 8 KB transfer on a current GbE switch) is also significant in a SAN. Application performance degrades with increasing latency to remote storage, but fortunately, end-system optimizations such as caching (see Section 4.4) and synchronous I/O suppression (see Section 4.3.1) often can hide network latency. In a lossy environment such as the Internet, bandwidth impairment is the dominant limitation on remote storage performance, so we may need new network techniques, e.g., to avoid the bandwidth destruction that TCP suffers under conditions of congestion and packet loss.

4.3.3 Remote storage

Slower storage system = better delay tolerance: If the remote storage device is slow, that bottleneck reduces the impact of network latency. This is seen in Figure 4, which compares the throughput of the SDET benchmark on remote storage using either a single IBM disk, or a disk array with 8 disks. The performance on the remote disk array is much more sensitive to network delay than is the single-disk performance.

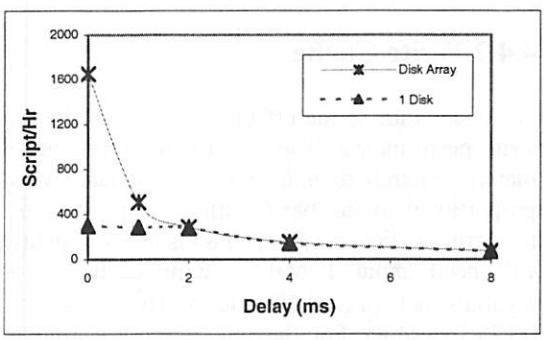


Figure 4: Effects of storage system on SDET performance on FreeBSD and Soft Update file systems.

4.4 Using cache to hide network delay

In this section we study the extent to which caches at the HG or the SG can hide network latency. We assume that we have reliable main memory or NVRAM [17] so that writes can be safely acknowledged from the cache, and we have no cache coherency issues because the

remote storage is partitioned over hosts that have exclusive access to virtual volumes.

4.4.1 Cache Location

A cache at the HG tends to hide both disk and network latency. We see this in Figure 5, which uses the 500 MB working set PostMark benchmark on FreeBSD to show the effectiveness of a 1 MB cache that is located at the HG, by comparison with a cache at the SG, or no cache (the latter two curves coincide). We see that the cache at the host side of the network can hide almost all of the network latency.

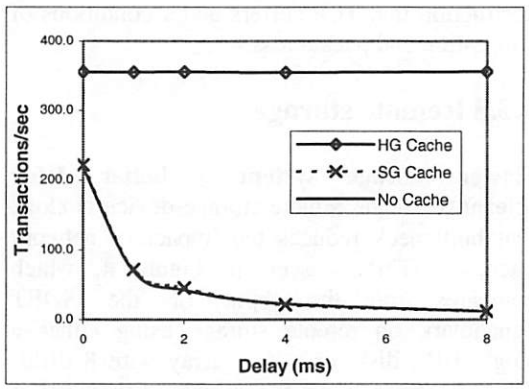


Figure 5: Performance of PostMark with 1 MB cache in HG or SG, versus network delay in ms. (FreeBSD, Soft Updates, delay testbed.)

4.4.2 Write Cache

We now examine the effect of the HG cache on write performance. The size of the write cache that is required to hide the network latency is proportional to the bandwidth-delay product of the network. For instance, an 8 ms network delay will need about 1 MB of write cache on a gigabit/s network (0.008 sec $\times$ 10^9 bits/sec $\times$ 0.125 bytes/bit). For the Internet, with variable delay and congestion, we may need a significantly larger cache.

Figure 6 shows the performance of PostMark running on FreeBSD with the Soft Updates file system on the congestion testbed. We show curves for 4 different HG cache sizes, where the size is given in the legend as a percentage of the 500 MB PostMark working set size. The baseline configuration's performance (no cache in Figure 6) is initially latency bound, due to the 0.46 ms round-trip delay inherent in the router network. It becomes bandwidth bound when the background load exceeds 55% of the backbone link. We can overcome the 0.46 ms router latency with a small amount of cache (i.e., 0.1% or 256 KB). This enables the application performance to approach that of a locally-attached disk array. For the 0.1% cache configuration, PostMark becomes bandwidth bound when the background traffic exceeds 30% of the backbone link. Its performance approaches that of the baseline "no cache" configuration. As we increase the cache size to 10% of the working set, PostMark is able to maintain a high transaction rate up to a 55% load factor. This is because the larger cache filters write traffic effectively, reducing it by 40%.

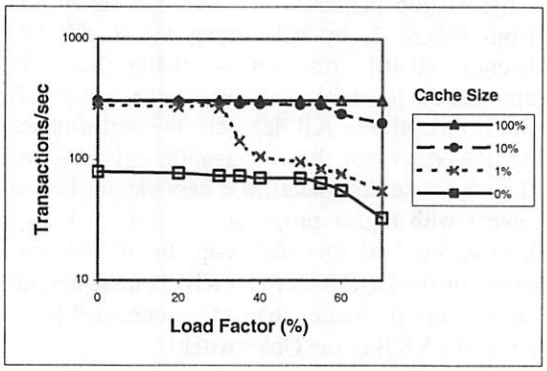


Figure 6: Impact of HG cache size on PostMark (FreeBSD, Soft Updates, congestion testbed; HG cache size specified as a percentage of the 500 MB PostMark working set size.)

4.4.3 Read Cache

Recall from Section 4.3.1 that NT has a very small file system buffer cache, and thus it incurs many capacity read misses. We now use the PostMark measurements on NTFS to examine the effectiveness of the HG read cache.

Figure 7 shows that the HG cache significantly improves performance under all load and delay conditions. We see that a large cache is needed to achieve performance similar to that of a locally attached disk array. For example, in the congestion test we see that with no background traffic and no cache, the throughput is 50 transactions per second. To tolerate a 70% loaded backbone with similar performance, the required cache size is 30% of PostMark's working set. Similarly, the delay test shows that 60% of the working set needs to be cached to mask a one-way propagation delay of 8 ms.

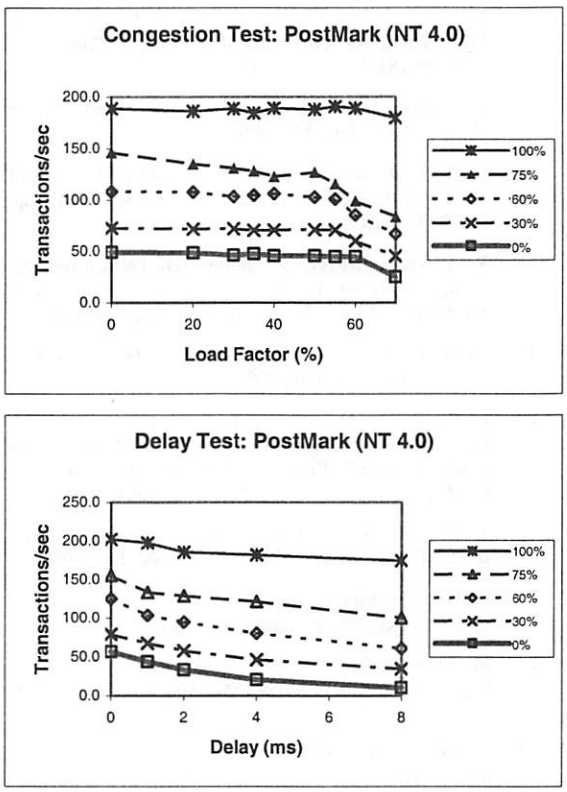


Figure 7: Effect of cache size on PostMark performance in the congestion and delay testbeds. Windows NT Server 4.0, NTFS, with HG cache size specified as a percentage of the 500 MB PostMark working set size. Performance is expressed in terms of Throughput and is based on an average of five runs. The standard deviation is less than 2.5% of the mean, with a maximum deviation of 3.9%.

5 Concluding Remarks

Storage outsourcing is an important emerging industry that forces us to revisit problems at the nexus of storage systems, file systems, and network protocols.

In this paper, we report measurements of the performance of remote storage via extensive benchmark experiments, using benchmarks that are widely applied in file system and database research. Our measurements cover FreeBSD with two versions of FFS, and Windows NT and Windows 2000 with NTFS and the FAT file system. We consider the impact of network delays when a host accesses remote storage over local or long-haul networks, and we also study the effect of congestion and packet loss on the performance of remote storage.

Our measurements give quantitative confirmation, in a SCSI over IP remote storage setting, of the dominant importance of write traffic, the benefits of suppressing synchronous writes, the significance of the file system buffer cache design, and the effectiveness of host-side caches.

Network delay can adversely affect application performance. But it can be masked in several ways, such as by write caching and read prefetching. The impact of network delay on performance is strongly influenced by the remote storage architecture and the network protocol settings. Slow remote storage masks network delay. In the case of fast remote storage, judicious caching and prefetching strategies are important tools to reduce the sensitivity of application performance to network delay.

We have observed quite acceptable performance for application benchmarks that access remote storage, even given network propagation delays that correspond to distances of hundreds of kilometers — but only in the absence of congestion and packet loss. An important problem for future work is to develop network and file system techniques that enable remote storage systems to maintain good performance when faced with the delay and loss characteristics of the Internet. In particular, to maintain high application performance despite high network latency, we need techniques to keep the network connection full of requests and data. Examples of techniques that can help to achieve high I/O concurrency include asynchronous I/O, multithreading, informed prefetching, and set-oriented data access methods. To maintain acceptable application performance in conditions of congestion and loss, we must revisit some traditional problems in networks having a high bandwidth-delay product. TCP bandwidth suffers from an insufficient ability to distinguish between loss and congestion, so SCSI over IP needs assistance, such as from QoS mechanisms or from other techniques that maintain high bandwidth despite loss. Many additional problems also remain for future work. For instance, small blocking I/Os for metadata updates can cause serious performance problems under high network delay. Can the file system maintain safety in a less costly way? Can the OS buffer cache do a better job by combining its write traffic into large scatter-I/O commands?

Acknowledgments

We thank our shepherd, Rod Van Meter, for his careful reading and valuable feedback. We thank the anonymous reviewers for their detailed comments. We thank our colleague Hao Sun for helping to set up the congestion testbed.

References

1 P. Barford, M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", *Proc. ACM SIGMETRICS*, pp. 151-160, Madison, WI, June 1998.

2 L. Brakmo, S. O'Malley, L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance", *Proc. ACM SIGCOMM*, pp. 24-35, London, UK, August 1994.

3 Chaparral Network Storage Inc, *TRC-22 User's Guide*, November 2000.

4 K. Claffy, "The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone", *Proc. Inet '98*, http://www.caida.org/outreach/papers/Inet98.

5 P. Danzig, S. Jamin, "tcplib: A Library of TCP Internetwork Traffic Characteristics", Technical Report CS-SYS-91-495, Computer Science Department, USC, 1991.

6 EMC, *Symmetrix Remote Data Facility Product Description Guide*, 2000.

7 S. Gaede, "Perspectives on the SPEC SDET Benchmarks", http://www.spec.org/osg/sdm91/sdet/index.html.

8 G. Ganger, Y. Patt, "Metadata Update Performance in File Systems", *Proc. 1st USENIX Symp on OSDI.*, pp. 49-60, Monterey, CA, November, 1994.

9 G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka, "A Cost-Effective, High-Bandwidth Storage Architecture", *Proc. 8th Int. Conf. ASPLOS*, pp. 92-103, San Jose, CA, Oct., 1998.

10 IBM Corp., *Ultrastar 36LZX Hard Disk Specifications, Version 1.0 (Revised)*, December 1999.

11 InterOperability Lab, *iSCSI Implementations*, http://www.iol.unh.edu/consortiums/index.html.

12 J. Katcher, "PostMark: A New File System Benchmark", Technical Report TR3022. Network Appliance Inc, October 1997.

13 T.V. Lakshman, U. Madhow, "The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss", *IEEE/ACM Trans. Net.* 5(3), pp. 336-350, June 1997.

14 R. Martin, D. Culler, "NFS Sensitivity to High Performance Networks", *Proc. ACM SIGMETRICS*, pp. 71-82, Atlanta, GA, May 1999.

15 M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for Unix", *ACM Trans. On Comp. Sys.* 2(3), pp. 181-197, August 1984.

16 F. Neema, D. Waid, "Data Storage Trend", *UNIX Review*, 17(7), June 15, 1999.

17 W. T. Ng, P. M. Chen, "The Design and Verification of the Rio File Cache", *IEEE Trans. Comp.* 50(4), pp. 322-337, April 2001.

18 W. T. Ng, B. Hillyer, "Obtaining High Performance for Storage Outsourcing (Poster Paper)", *Proc. ACM SIGMETRICS*, pp. 322-324, Boston, MA, June 2001.

19 Oracle Corp. *Oracle8i Administrator's Guide, Release 2 (8.1.6)*, pp. 3-31, Jan 2000.

20 B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, D. Hitz, "NFS Version 3 Design and Implementation", *Proc. USENIX Summer 1994 Tech. Conf.*, pp. 137-152, Boston, MA, June 1994.

21 V. Paxson, "End-to-End Internet Packet Dynamics", *IEEE/ACM Trans. Net.*,7(3), pp. 277-292, June 1999.

22 L. Rizzo, "dummynet", http://info.iet.unipi.it/~luigi/ip_dummynet.

23 M. Russinovich, *CacheSet v1.0*, Dec. 1998, http://www.sysinternals.com/ntw2k/source/cacheset.shtml.

24 J. Satran, et. al., "iSCSI", Sep. 2001, http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-08.txt.

25 M. Seltzer, G. Ganger, M. McKusick, K. Smith, C. Soules, C. Stein, "Journaling versus Soft Updates: Asynchronous Metadata Protection in File Systems", *Proc. USENIX Annual Tech. Conf.*, pp. 71-84, San Diego, CA, June 2000.

26 S. Shepler, et. al., "RFC 3010: NFS Version 4 Protocol", Dec. 2000, ftp://ftp.isi.edu/in-notes/rfc3010.txt.

27 A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill, 1998.

28 D. Solomon, *Inside Windows NT Second Edition.* Microsoft Press, 1998.

29 C.A. Thekkath, T. Mann, E.K. Lee "Frangipani: A Scalable Distributed File System", *Proc. 16th ACM SOSP*, pp. 224-237, San-Malo, France, Oct., 1997.

30 K. Thompson, G. Miller, R. Wilder, "Wide Area Internet Traffic Patterns and Characteristics", *IEEE Net.*, 11(6), Nov.-Dec. 1997.

31 Transaction Processing Performance Council, *TPC Benchmark C, Standard Specification, Revision 5.0.* February 2001, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

32 R. Van Meter, G. G. Finn, S. Hotz, "VISA: Netstation's Virtual Internet SCSI Adapter", *Proc. 8th Int. Conf. ASPLOS*, pp. 71-80, San Jose, CA, Oct., 1998.

33 R. Wang, T. Anderson, D. Patterson, "Virtual Log Based File Systems for a Programmable Disk", *Proc. 3rd USENIX Symp on OSDI.*, pp. 29-44, New Orleans, LA, February, 1999.

PersonalRAID: Mobile Storage for Distributed and Disconnected Computers

Sumeet Sobti* Nitin Garg* Chi Zhang* Xiang Yu*
Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

This paper presents the design and implementation of a mobile storage system called a PersonalRAID. PersonalRAID manages a number of disconnected storage devices. At the heart of a PersonalRAID system is a mobile storage device that transparently propagates data to ensure eventual consistency. Using this mobile device, a PersonalRAID provides the abstraction of a single coherent storage name space that is available everywhere, and it ensures reliability by maintaining data redundancy on a number of storage devices. One central aspect of the PersonalRAID design is that the entire storage system consists solely of a collection of storage logs; the log-structured design not only provides an efficient means for update propagation, but also allows efficient direct I/O accesses to the logs without incurring unnecessary log replay delays. The PersonalRAID prototype demonstrates that the system provides the desired transparency and reliability functionalities without imposing any serious performance penalty on a mobile storage user.

1 Introduction

As disk density continues to grow at a phenomenal annual rate of 100% [3], the cost, form factor, and capacity of stable storage continues to improve dramatically. One consequence of these dramatic technological advances is the emergence of highly compact secondary storage, which can be seamlessly integrated into devices of all shapes and forms. As technology continues to improve, as decentralization is carried to its logical next step, and as traditionally analog information is increasingly being turned into digital representations, it is not unreasonable to conjecture that such mobile storage may become a dominant form of storage in the near future, especially for personal user data, subsuming conventional disks enshrined in machine rooms.

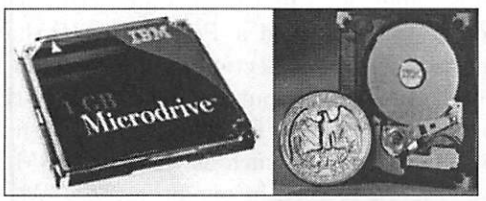


Figure 1: An IBM Microdrive. On the left is a 1 GB Microdrive shown in its packaging. On the right is the same drive shown open with a U.S. quarter. (Courtesy of IBM. Unauthorized use not permitted.)

Unfortunately, as mobile storage flourishes, high-performance universal network connectivity may still not be available everywhere. At any instant, only a small number of devices may be connected to each other; and a mobile storage user cannot always count on an omnipresent high-quality connectivity to a centralized storage service. A mobile storage solution that does not rely solely on network connectivity for managing a collection of distributed (and possibly disconnected) devices needs to be found. We also share the belief that a user's attention is a precious resource that the system must carefully optimize for, and a central goal of the system is to ease the management of these disconnected devices.

We identify three important desirable features of such a mobile storage solution. (1) *The availability of a single coherent name space.* A user who owns a number of storage devices should not be burdened with the chore of hoarding needed data and propagating updates manually. Ideally, even when these devices are permanently disconnected, the user still sees a single coherent space of data regardless where she is and regardless which device she uses. The user should not have to modify her existing applications to enjoy these benefits. (2) *Reliability.* The view that only centralized servers provide reliability guarantees and mobile devices are inferior second-class citizens, whose data is expendable, is not al-

*Department of Computer Science, Princeton University, {sobti,nitin,chizhang,xyu,rywang}@cs.princeton.edu.
†Department of Computer Science, Yale University, arvind@cs.yale.edu.

This work is supported in part by IBM. Wang is supported by NSF Career Award CCR-9984790 and Krishnamurthy is supported by NSF Career Award CCR-9985304.

ways acceptable: the frequency and duration of periods where a mobile storage device is disconnected from a central server can be significant enough that one must have some degree of confidence over the reliability of data. (3) *Acceptable performance.* The provision of the transparency and reliability features listed above should not impose a significant overhead. Ideally, the user of the mobile storage system should always be able to enjoy a level of performance that is close to that of the local storage.

In this paper, as a first step, we describe a mobile storage system called a *PersonalRAID* that is designed to support a collection of disconnected and distributed personal computers. In addition to these end hosts, central to the PersonalRAID design is a portable storage device, such as the 1 GB IBM Microdrive (Figure 1) used in our prototype. We call this device the *Virtual-A* (VA). The VA is named so because: (1) it is analogous to a conventional removable storage device (which is often named drive A on Windows PCs), and (2) it provides the illusion of a storage device whose capacity is far greater than the device's physical capacity. The VA allows the PersonalRAID system to achieve the three goals enumerated above: (1) The VA transparently propagates updates among disconnected hosts to ensure eventual consistency. It helps provide a single storage name space that is transparently available on all hosts. It supports existing file systems and applications without modification. (2) The VA provides temporary redundancy before data is propagated to multiple end hosts so that the PersonalRAID system can tolerate any single-device loss. (3) The PersonalRAID system intelligently chooses between a local disk and the VA device to satisfy I/O requests to mask propagation delays and minimize overhead.

The central aspect of PersonalRAID design is its use of a distributed log-structured design: the collection of distributed logs *is* the storage system: there is no other permanent structure hosting the data. This design not only allows PersonalRAID to propagate updates among the logs throughout the system efficiently, it also allows a user to satisfy her I/O requests directly from the logs without having to wait for propagations to complete. We have implemented a prototype PersonalRAID system. Our experiments demonstrate that the system achieves the transparency and reliability functional goals without imposing any serious performance penalty.

The rest of the paper is structured as follows. Section 2 describes the user experience and the main operations of a PersonalRAID system. Section 3 presents the rationale and the details of the log-structured design of the system. Section 4 describes our prototype PersonalRAID implementation. Section 5 details the experimental results. Section 6 compares PersonalRAID to a number of related systems. Section 7 concludes.

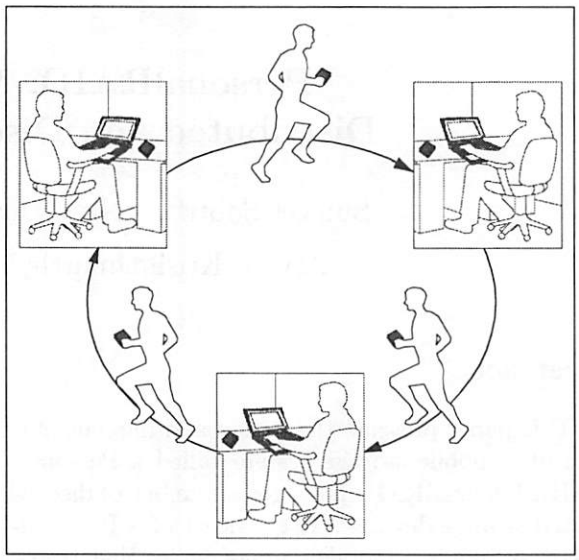
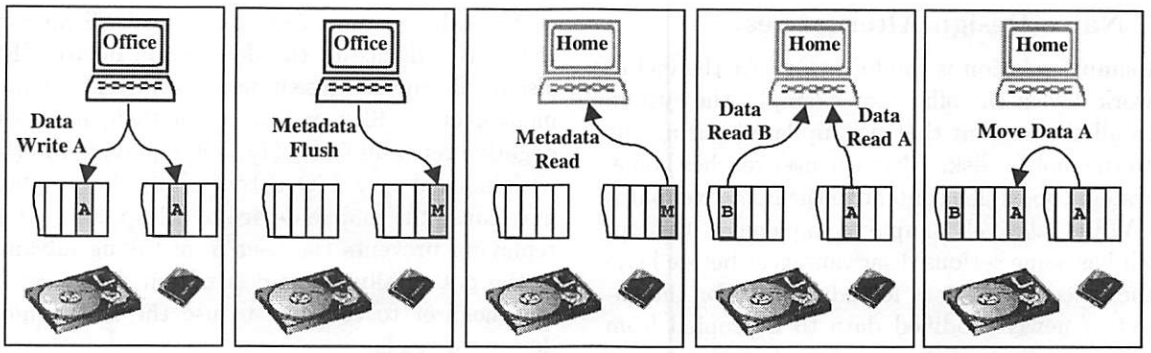


Figure 2: A usage scenario of PersonalRAID. The mobile user carries the Virtual-A device as the user travels among disconnected computers (such as a machine at home, a machine in the office, and a laptop on the go). The PersonalRAID system provides the illusion of a single name space, and it also ensures data reliability.

2 Functionalities

The PersonalRAID system manages a number of disconnected storage devices where the mobile user desires a single name space on all of them. The mobile VA device is instrumental in bringing this about. It is generally not a good idea to rely exclusively on the mobile storage device alone due to its capacity, performance, and reliability limitations; instead, the device needs to be an integral part of a PersonalRAID system.

The VA accompanies the user wherever she goes (Figure 2). With current technology, a few gigabytes can be packaged in the form factor of a credit card (such as the Kingston 5 GB DataPak PC Card Type II Hard Drive) or a wrist watch (such as the IBM 1 GB Microdrive). The VA can communicate with a host computer via various forms of connectivity (such as PCMCIA, USB, or Bluetooth). As long as the VA is present, the user "sees" her up-to-date large home directory regardless where she is and which computer she is using. The user never needs to perform manual hoarding or manual propagation of data; and the loss or theft of any single device does not result in data loss. PersonalRAID mainly targets personal usage scenarios. (We ad-



(a) Record (b) Disconnect (c) Connect (d) Read (e) Replay

Figure 3: PersonalRAID operations. The roles of the home and office computer disks are reversed as the user creates new data at home, which must be propagated to the office.

dress concurrent updates in Section 3.8.1.)

Figure 3 depicts in greater detail the PersonalRAID operations that synchronize the contents of several host computer disks. (a) In this example, when in the office, the VA passively observes the I/Os performed to the office computer disk and incrementally *records* the newly written data. The office computer disk is called the *source* of this data. (b) When the user is about to leave the office and *disconnects* the VA, the system flushes some metadata so that an inventory of the VA's contents is placed on it. (c) The user then takes the VA home and *connects* the device to her home computer. The system reads the metadata, and file system operations can occur immediately following connection. (d) After connection, the system *reads* from either the VA or the home computer disk to satisfy user requests. We call the I/O events that have occured between a pair of connection and disconnection events a *session.* (e) Possibly in the background, PersonalRAID synchronizes the contents of the disks by *replaying* some of the updates, which were recorded on the VA earlier in the office, to the home computer disk. The home computer disk is called the *destination.* Only after the latest updates are reflected on all the host disks do we remove the copy of the data from VA. (This invariant can be a problem if some hosts in the system are only infrequently visited by the user, and the VA device is not large enough to hold all the unpropagated data.) Note that it is not necessary to replay all the new data to a destination device in a single session—the user may choose to disconnect from the home computer at any time. As the user creates new data at home and the VA records it for later replaying to the office disk, the roles of the two end hosts are reversed: the home disk becomes the source and the office disk becomes the destination. Note that we are maintaining an invariant: a copy of any data resides on at least two devices. This invariant allows the system to recover from any single-device loss.

Our current implementation requires the VA device to be present when the file system is being accessed; this is consistent with the most common single-user case that PersonalRAID targets. We do not currently handle updates on a host that is disconnected from the VA. Such updates can be potentially conflicting. In Section 3.8.1, we discuss ways in which the current system could be extended to address these limitations. Another limitation is that the incorporation of a new host into the system requires a heavy-weight recovery operation, as discussed in Section 3.6.

For simplicity, our current system addresses only disconnected computers. We are researching extensions of the system that can exploit weak or intermittent network connectivity when it is present. We expect the mobile storage device and the weak connectivity to complement each other in such situations.

3 Design

The design of the PersonalRAID should satisfy the following requirements:

- Recording should not impose excessive overhead that may interfere with normal I/O operations.
- During disconnection, the user should not be forced to wait for long before the VA can be safely removed.
- During connection, the user should not be forced to wait for long before she is allowed to perform I/O operations.
- Replaying should not impose excessive overhead that may interfere with normal I/O operations.
- Replaying should proceed quickly so that the disk space on the VA can be quickly freed up for future I/Os.

3.1 Naive Design Alternatives

A simple solution is the following. At the end of the work day in the office, for example, the system copies all the content that was updated during the day to the mobile disk. After the user reaches home, the system copies this content to the home computer disk. While relatively simple to implement, this approach has some serious disadvantages: before leaving the office, the user is forced to wait for the entire set of newly modified data to be copied from the source to the mobile device; and after reaching home, the user must wait for the entire mobile device content to be copied to the destination before she is allowed to access the file system. With mobile storage devices that can store gigabytes of data, and with potentially lengthy intervals spent at one computer before moving onto another, the latency may become intolerable.

One possible improvement is to incrementally copy newly generated data from the source disk to the mobile disk in the background instead of allowing the new data to accumulate. This improves the disconnection time, but it does not address the long connection latency–the user still must wait for the entire propagation to complete before she can proceed to normal I/O activities.

To address these disadvantages, one realizes that the PersonalRAID needs to be a file system or storage system solution which can transparently decide which device to access for a particular piece of data: this is necessary, for example, if the user desires to access the data on the mobile device after connection but before it is propagated to the destination device. With this requirement in mind, let us consider a second alternative: during recording, the system mirrors a portion of the source device Unix File System (UFS) on the mobile device. After connection, while background replaying occurs, the user may transparently access the mirrored portion of the UFS on the mobile device.

While this second design alternative may improve the disconnection and connection latency, the choice of mirroring a portion of the UFS on the mobile device may not be a wise decision for best recording and replaying performance. First, during recording, UFS updates may incur a large number of small synchronous disk writes. The problem is made worse when the mobile storage devices typically do not possess the best latency characteristics so it is difficult to mask the extra latency by overlapping the I/Os to different devices. This situation is especially unfortunate when one realizes that the synchronous mirroring on the mobile device is unnecessary–data is already made persistent on the source device.

Second, replaying from a partial UFS mirror on the mobile device to the destination device UFS is also inefficient, because unnecessary disk head movement occurs. Slow replaying, in turn, has several negative repercussions: (1) slow replaying interferes with normal user I/O activity; (2) slow replaying may cause the mobile device to fill up; and (3) slow replaying prevents the user from taking advantage of the potentially faster destination device by forcing the user to continue to use the slower mobile device for reads.

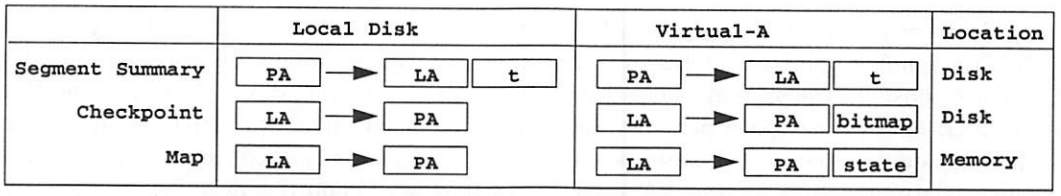
3.2 Log-Structured Organization

The analysis of the naive design alternatives shows that a good PersonalRAID design should have at least these two properties: (1) the mobile device should be an integral part of a storage or file system so that the user can transparently read/write the device without incurring long connection/disconnection latency; (2) the transfer of data onto/off the mobile device should take place in a fashion that avoids incurring the intrinsic latency bottleneck of disks.

One possible design that naturally satisfies these requirements is to have some variant of a log-structured file system (LFS) [12] on both the VA and the host disks. During recording, data is buffered in large memory *segments*; these memory buffers prevent overwritten data from ever reaching the disks and large segment-sized writes are efficient. Disconnection is analogous to a graceful LFS shutdown and connection is analogous to an LFS recovery, both of which mainly involve metadata operations that are relatively efficient. Note that disconnection must also flush any dirty data segments. Fast replaying is possible because the system transfers data at large segment-sized granularity that fully utilizes the VA and host disk bandwidth. Furthermore, during replaying, as the system reads live data from the VA and writes them to the destination device, large extents of empty segments are generated on both the VA and the destination device; therefore, replaying and segment cleaning in effect become an integral one.

3.3 PersonalRAID Data Structures

While it is possible to build a PersonalRAID system at the file system level by modifying an LFS to adapt to multiple storage devices, we have elected to construct the PersonalRAID by extending the design of a Log-Structured Logical Disk (LLD) [1]. A *logical disk* behaves just like a normal disk from the point of view of a file system: it allows the file system to read and write *logical disk addresses*. A particular



PA	physical device address.
LA	logical (virtual) device address.
t	time stamp (global counter).
bitmap	one bit (b_i) per host, b_i=1 if host i needs propagation.
state	4 bits of state information for a block.
s_0	1 iff the block needed to be propagated to the current host at the beginning of the current session.
s_1	1 iff the block needed to be propagated to any of the other hosts at the beginning of the current session.
s_2	1 iff the block has been propagated from the VA to the current host in the current session.
s_3	1 iff the block has been overwritten in the current session.

Figure 4: Details of the main PersonalRAID data structures.

implementer of a logical disk, however, can choose to map these logical addresses to physical addresses in a way that she sees fit. A log-structured logical disk maps logical addresses that are written together to consecutive physical addresses, effectively accomplishing the goals of an LFS. An LLD can support existing file systems with little or no modification and it is typically easier to implement an LLD than an LFS. The PersonalRAID system makes a single consistent logical disk available on all the participating hosts, despite the lack of *any* network connection between any of them.

The data structures on each device of a PersonalRAID are not unlike those of a conventional LLD. In a conventional LLD, the disk is structured as a segmented log. Data blocks are appended to the log as the LLD receives write requests from the file system. Each write request is assigned a *unique* time stamp from a monotonically-increasing global counter. Each segment contains a segment summary, which has the logical address and the time stamp of each data block in the segment. The segment summary aids crash recovery of the in-memory logical-to-physical (L-to-P) address mapping. The L-to-P mapping is checkpointed to the disk during graceful shutdown.

Figure 4 shows in greater detail the three main PersonalRAID data structures: the *segment summary*, the *checkpoint*, and the *in-memory map*. For each of the three data structures, there is one version for the host local disk and there is another version for the VA. The version for the local disk is essentially the same as that of a conventional LLD. The VA version is augmented with some additional information.

Each entry of the L-to-P mapping in the VA checkpoint is augmented with a bitmap: one bit (b_i) per host, and $b_i = 1$ if the block for this logical address needs to be propagated to host i. In a PersonalRAID system, the contents of the VA device are *defined* to be the set of data blocks that still need to be propagated to some host. Thus, a block is evicted from the VA device when it has reached each host in the system. In other words, the L-to-P entry for a particular logical address is marked null when the corresponding bitmap contains 0 in all positions. The VA checkpoint is also home to a global counter. All write operations in the system are assigned a unique time stamp from this counter.

Each entry of the VA in-memory map is augmented with a state field that consists of four bits (s_{0-3}): this field is a summary of what *needed* to happen to this block at the beginning of the current session in terms of propagation (s_{0-1}), and what *has* happened to this block on this host in the current session in terms of replaying and recording (s_{2-3}).

When stored as a table, each L-to-P map consumes 4 bytes per logical address. Thus, assuming a block size of 4 KB, the L-to-P map needs 1 MB for each GB of logical address space. The total size

of the bitmap fields in the VA checkpoint is MN bits, where N is the number of logical addresses and M is the number of hosts in the system. Thus, the bitmaps need 32KB of VA disk space per host for each GB of logical address space. The state fields in the in-memory data structure for the VA need 128KB for each GB of logical address space. Observe, however, that at any time, most L-to-P entries and the state and the bitmap fields for the VA will be null, since the size of the VA will typically be much smaller than the size of the logical address space. Thus, more compact representations of the VA data structures are possible.

We close this section by making the following observation about the PersonalRAID data structure design. Each device in a PersonalRAID is self-contained in that the physical addresses within the data structures for this device all point to locations within this device. If a logical data block is not found on this device, the pointer to it in the L-to-P mapping of this device is null; in this case, a corresponding pointer on a device that does contain the block points to a true location of the block. The union of all the valid pointers on all devices constitutes the whole logical-to-physical map.

3.4 PersonalRAID Operations

In this section, we describe the various PersonalRAID operations in detail. Figure 3 also illustrates how these operations interact with the underlying log on each device.

3.4.1 Recording

During recording (a), the system appends a newly created logical block to two logs: one on the source device and the other on the VA. The logical address of the newly created data is recorded in each of the two segment summaries along with the latest timestamp. The in-memory map of each device is also updated to reflect the latest locations of the data block. In addition, we set $s_3 = 1$ in the state field of the VA in-memory map to mark the creation event during this session.

3.4.2 Disconnection and Crash Recovery

During disconnection (b), first the file system on the PersonalRAID logical disk is unmounted. Unmounting flushes all the dirty file-system buffers to the PersonalRAID logical disk, and it also indicates to the user that the file system on the local host is unusable when the VA is not connected to it. Then, a graceful shutdown is performed on both the local disk and the VA. For the local disk, we simply write the in-memory map to its checkpoint region. (If the local host is not powering down after disconnection, then the system can choose to keep the map in memory—this optimization can reduce the connection time for the next session on this host. See Section 3.4.3.)

The VA checkpoint region contains the VA L-to-P map and the bitmap fields. Thus, for the VA, in addition to flushing the map, we must read the bitmap fields of the old checkpoint into memory, compute the new bitmap fields using the old bitmap fields and the state fields of the in-memory map, and write a new checkpoint back to the VA. If $s_3 = 1$ (*written* block), we set the bitmap to reflect the need of propagating this block to all other hosts in the system. Otherwise, if $s_2 = 1$ (*propagated* block), we clear the corresponding bit for this host in the bitmap but retain the values of the remaining bits. We also store the latest timestamp in the VA checkpoint; this timestamp marks the end of the current session and the beginning of the next session. To avoid corrupting the old checkpoint in case a crash occurs in the middle of a checkpoint operation, we maintain two checkpoint regions for each device and alternate between them.

A crash is a special case of disconnection. In a conventional LLD, the goal of crash recovery is to reconcile the contents of the segment summaries and the checkpoint to make them consistent with each other. In the PersonalRAID system, however, the crash recovery process also needs to make the local disk and the VA *mutually consistent* and to restore the PersonalRAID invariants. Note that the PersonalRAID system requires that the recovery process be completed at the crash site before the user moves to another host.

The first PersonalRAID invariant to restore is that all data blocks written in the past (unfinished) session must be present on both the local disk and the VA disk. Because the flushes to the two disks are not synchronized, one of them might have fallen behind the other in terms of receiving the most recent writes. Thus, the crash recovery process might need to propagate data blocks from one disk to the other.

The other PersonalRAID invariant to restore is that the bitmaps in the VA checkpoint must correctly reflect the state of the system in terms of propagation of writes. Blocks written or propagated during the past session might have made the old bitmaps inconsistent. Such blocks can be identified by comparing the time stamps in the segment summaries with the time stamp in the old VA checkpoint. Having identified the written and propagated

blocks, bitmaps are updated in the VA checkpoint as in the case of a normal disconnection described above. If all bits in a bitmap are clear, we must have performed all the necessary propagations and we can safely discard this block from the VA by nullifying the logical-to-physical mapping for this block on the VA. We also store the latest timestamp found during the segment summary scan in the VA checkpoint to mark the end of this session.

3.4.3 Connection and Reading

During connection (c), the system needs to initialize the in-memory maps by reading the checkpoints. If the host that the VA is connecting to is powering up, the system needs to initialize the local disk in-memory map by reading the local disk checkpoint, just as LLD does. In a similar fashion, the system reads the VA checkpoint to initialize the VA in-memory map. The difference between the two maps is that the system also needs to calculate the state field of the in-memory map for the VA: s_0 and s_1 are set based on the bitmap stored in the checkpoint (s_0 is set equal to the current host's bit and s_1 is set to the inclusive OR of all other bits), while s_2 and s_3 are cleared. And if $s_0 = 1$, the system concludes that the local disk contains an obsolete copy, and nullifies the logical-to-physical mapping in the local disk in-memory map. Finally, the system also reads the disconnection timestamp in the VA checkpoint to initialize the current timestamp.

As soon as connection completes, the system is ready to accept I/O requests. To service a read request (d), the PersonalRAID looks up the in-memory maps for valid logical-to-physical mappings to decide which device holds the most recent copy of a logical block. In the event that a fresh copy resides on multiple devices, the system is likely to favor the local disk, which is typically faster than the VA, although load-balancing opportunities exist. To service a write request, the PersonalRAID records the new block by appending it to both logs as described earlier.

3.4.4 Replaying

Possibly in the background, the system performs replaying (e). A great deal of synergy exists between PersonalRAID and log-structured storage as replaying is integrated with segment cleaning on the VA device. The system checks the s_0 bit in the VA in-memory map to identify the live data on the VA that is yet to be propagated to the destination device. It then reads live data from the VA and appends it to the log on the destination device. The timestamp of the newly propagated block inherits that of the VA block, which the system reads from the corresponding VA segment summary. The s_2 bit is set in the state field. Next, the system checks s_1 to determine whether this block needs to be propagated to other hosts in the system. If $s_1 = 0$, the data block must have been propagated to all hosts in the system and the block can be safely removed from the VA. If we attempt to populate a segment with blocks of identical propagation bitmaps, then we are likely to harvest free segments as their blocks are freed simultaneously. On the other hand, if $s_1 = 1$, we must retain the block on the VA for further propagation to other hosts in the system and we have two options: we can either leave the blocks in place on the VA in the hope that they may be deleted in the future to render their cleaning unnecessary, or append them to the end of the VA log since we have already incurred the cost of reading them into memory. In the latter case, we have again accomplished segment cleaning on the VA as a byproduct of replaying.

In a conventional LLD, the segment cleaning algorithms mainly aim to maximize the number of free segments generated per unit of cleaning I/O. This is typically achieved by cleaning segments that are relatively cold and have low utilization [12]. (Recall that a piece of data is said to be *cold* if it is unlikely to be overwritten in the near future, and *utilization* of a segment is a measure of the amount of live data in it.) In a PersonalRAID system, however, segment cleaning on the VA device becomes more complicated due to its integration with replaying. Instead of (or in addition to) coldness and utilization of segments, a good cleaning strategy might need to take into account the state of propagation of blocks. For example, at any time, a live block on the VA might need to be propagated (1) to the local host, but not to any remote host, or (2) to some remote host, but not to the local host, or (3) to the local host as well as some remote host. A cleaning strategy might prefer to clean one type of blocks before others depending on how aggressively it wants to replay blocks or generate free segments.

In this section, we examine the log operations during the recording, disconnection, connection, reading, and replaying phases of the PersonalRAID. We note two features of the PersonalRAID design. One feature is that the map information on each device is self-contained: all the valid pointers in a device point to locations within the same device. This feature allows data movement within one device or across a subset of the partially connected devices to be performed independently of other devices. The second feature of the PersonalRAID design is the

potential for exploiting I/O parallelism: the recording, reading, and replaying phases can overlap I/Os to the VA and local disks to mask some of the I/O latency and balance load.

3.5 Recovering from Device Losses

We first discuss how to recover from the loss of a host computer disk. Then we discuss how to recover from the loss of the VA device itself.

3.5.1 Recovering from Host Disk Loss

This scenario is the simpler case to handle. We take the VA to a surviving host. The union of the contents of the local disk on this host and the VA gives the entire content of the PersonalRAID. First, we completely synchronize the local disk and the VA by replaying all those blocks whose latest versions are present on the VA but not on the local disk. Thus, at the end of this phase, the bit for this host in all the bitmaps is 0. Then, we create a physical mirror of the local disk onto a new disk (using a Unix utility like `dd`, for example). The new disk is brought to the accident site to replace the lost disk. The bitmaps on the VA are updated to reflect the fact that the restored disk now has all the PersonalRAID data.

3.5.2 Recovering from VA Device Loss

The more complex case is when the VA device itself is lost. There are two pieces to be reconstructed. The first piece is the metadata, which consists of the bitmaps and the current value of the global counter. The other is the set of actual data blocks that were lost with the VA device. Recall that the contents of the VA are *defined* to be the set of data blocks that must be propagated to some host in the system. Thus, to reconstruct the data part, we may need to visit all the hosts in the system.

A simple recovery method is this. We visit all the hosts in the system twice. The goal of the first tour is to construct the metadata part by scanning the segment summaries on each host and comparing time stamps. In the second tour, we simply visit each host to copy the required data blocks onto a new VA device. Note that this tour does not need to scan the segment summaries on the local disks—the L-to-P mapping is sufficient to locate the required data.

An unpleasant aspect of this reconstruction approach is that it requires one to visit each host twice. To eliminate the first tour, we can make a copy of the bitmaps and the global counter on the local disk during the disconnection process. When the VA is lost, the user must retrieve the bitmaps and the value of the global counter from the host where she most recently disconnected. The bitmaps allow the system to identify the hosts on which the latest copy of a VA logical block is stored. The user now needs to complete only one tour to reconstruct the VA content. The price one pays for this simpler approach is the extra time and space spent during disconnection to write the bitmaps to the local disk, although it is likely that one should be able to overlap the bitmap flush time on the local disk with the slower VA checkpoint time, and the space consumed by the bitmaps is insignificant.

This VA reconstruction approach even works if one encounters a combination of a crash and the loss of the VA. In this unfortunate scenario, the user retrieves the old bitmaps from the computer that she visited last and then comes back to the crashed computer to complete the crash recovery process as described in Section 3.4.2. At the end of this crash recovery process, the system has recovered the lost bitmaps and she can start the reconstruction tour to reconstruct the lost VA data. To avoid having to go back to the computer that she visited last in this scenario, we can make a copy of the VA checkpoint bitmaps on the local disk during the connection process (as well as the disconnection process).

The VA reconstruction processes described so far require the user to complete at least one tour of all the hosts. To further reduce the number of hosts that one must visit after a VA loss, one can periodically replay some blocks to *all* the hosts to eliminate these blocks from the VA or periodically replay *all* VA content to some host. The latter technique is effectively same as making a copy of the VA content on that host.

3.6 Reconfiguration

Reconfiguration of a PersonalRAID in the form of removing or adding hosts is relatively simple. To remove a host from the system, all the system has to do is to reformat the VA checkpoint to remove a bit from each bitmap. Recall that the bitmaps record which hosts in the system need to receive propagations. The checkpoint reformat allows the system to discard data from the VA, no longer propagating it to the removed host. To further simplify this process, the system can simply record at the beginning of the checkpoint which bits in the bitmaps are still considered *active*; so the non-active bits are not considered in the algorithm.

Adding a host to the PersonalRAID is essentially the same as recovering from the loss of a host disk. The only difference is the reformatting of the VA checkpoint to add a bit to each bitmap. Again, to

further simplify the process, the system can simply *activate* a previously allocated bit. This bit is set to 0 in each bitmap since at the end of recovery (see Section 3.5.1), the recovered host has all the PersonalRAID data.

3.7 Virtual VAs

So far in our discussion, we seem to have assumed that the Virtual-A has to be a physical mobile storage device such as the IBM Microdrive. This assumption is not necessary: the Virtual-A can in fact be backed by a file, a local disk partition, or even a network connection. We call such a virtual backing device a *Virtual Virtual-A* (or a VVA or a V^2A).

One possible use for a local disk-based VVA is to use it instead of a mobile storage device to perform recording. Because mobile storage devices do not necessarily have the best performance characteristics, recording to a VVA can be more efficient. Of course, to transport the data, we still must copy the contents of a VVA to a mobile device-based VA (and sometimes, vice versa). One possible solution for avoiding the long copying latency is to allow asynchronous copying to the mobile device to occur in the background. Although the log-structured design of the VA organization allows the PersonalRAID to buffer a large amount of data in memory and delete overwritten data before it reaches the device, the amount of buffering is still limited by the available memory. A VVA that asynchronously copies to a VA essentially allows unlimited buffering. Another use of a VVA is to make a copy of the VA for the purpose of reconstructing a lost VA (as described in Section 3.5.2).

To efficiently implement a local disk-backed VVA, one in fact does not need to physically and separately store the data blocks that can already be found on the local disk of this host: the mapping information is all that is needed. Because we do not have to physically store separate copies for a VVA's data in this case, we call this a *Virtual VVA* (or a V^3A).

3.8 Limitations and Extensions

In this section, we describe a number of limitations of the PersonalRAID system described so far. Some of these are the topics of our continued research and we discuss possible approaches of addressing them.

3.8.1 Concurrent Updates

To simplify the discussion so far, we have made a conscious design decision of not addressing concurrent updates. We believe that this is an acceptable choice for the use cases that we are targeting: PersonalRAID, being "personal", is designed for a single user to control a number of distributed and disconnected personal storage devices; these storage devices do not receive updates concurrently simply because we do not allow the user to be at multiple sites simultaneously. There is, however, nothing intrinsic in the current PersonalRAID design that prohibits us from addressing concurrent updates.

Indeed, there are legitimate personal use cases where concurrent updates arise naturally. For example, a networked office computer can continue to receive email after the user has disconnected the VA and has gone on vacation, taking a copy of the mail file with her. As long as the user does not modify the same mail file during the trip, the current PersonalRAID design can be trivially extended to accommodate such concurrent but non-conflicting updates.

After the user disconnects the VA, as the office computer receives new updates, it records the updates in a V^3A on the local disk (as described in Section 3.7). When the user returns and connects the mobile VA device to the office computer, as long as there is no conflicting update, the system can merge the checkpoint of the VA with that of the V^3A to arrive at a consistent VA image as a result of propagating the V^3A updates to the VA. After the merging, the system operates as described previously.

If there are conflicting updates, application or user-level intervention is necessary. In this case, a more sophisticated extension to the current PersonalRAID design is necessary. The logical disk approach upon which the PersonalRAID design is based becomes a convenient and powerful vehicle to support file system versioning. At VA disconnection time, the content of the local disk L-to-P map is "frozen" to represent a version (V_0). As updates are recorded in the local V^3A, none of the old blocks in V_0 are overwritten. The union of V_0 and the V^3A represents a new version V_1. After the user returns and connects her VA device, which may contain conflicting updates, the union of V_0 and the VA represents yet another version V_2. The user or an application must resolve conflicts to arrive at a "consistent" new version V_3. Upon conflict resolution, the old versions V_0, V_1, and V_2 can be freed and a consistent VA image again emerges.

3.8.2 Mobile Storage Limitations

Mobile storage technologies are likely to lag behind conventional ones in terms of performance and capacity. We believe that the performance disadvan-

tages are addressed by the log-structured design of the PersonalRAID and the judicious use of the host disk in the forms such as the V^3A described in Section 3.7. Currently, we do not address the capacity constraint. We are currently researching ways that a weak network connection (when one is present) may complement the mobile storage device to address this limitation.

3.8.3 Limitations of the Log-Structured Organization

The potential disadvantages of the log-structured organization are the possible destruction of read locality and the cost of segment cleaning (or disk garbage collection) [13, 14]. PersonalRAID mainly targets personal computing workloads, which are often bursty and leave ample idle time for cleaning. The cleaning overhead can be further reduced by using techniques like freeblock scheduling [7], and by buying bigger disks and keeping the disk utilization low.

The base LLD design that we have borrowed has several potential disadvantages [1]. Keeping the entire logical-to-physical map in memory, the base LLD design consumes a large amount of memory and incurs some latency when reading this map from disk into memory at startup time. It is, however, possible to cache only a portion of the map in memory and demand it in gradually. We have not implemented this possible optimization.

4 Implementation

In this section, we describe a Linux PersonalRAID implementation. As explained in Section 3, our system implements all the PersonalRAID functionalities at the logical disk level. The system consists of two main components: the PR Driver (PRD) and the PR Server (PRS) (shown in Figure 5). The PRD is a pseudo-block device driver that exports the interface of a disk. Upon receiving I/O requests, the PRD forwards them to the PRS via upcalls. The PRS is a user-space process that implements the LLD abstraction; it manages a partition on the local disk and the VA device in a log-structured manner as described in Section 3.

Most of the complexity in our system is concentrated in the PRS. Despite the upcall overhead, which is purely an implementation artifact, we chose this design for ease of programming, debugging and portability. Unfortunately, this decision also leads to some deadlock possibilities. A user process can cause the buffer cache to flush as a side effect of requesting system services (like dynamic memory allocation). The user process blocks till the flush is completed. If the PRS, which is just another user process, causes a flush to the PRD, the system enters a deadlock with the PRD waiting for the PRS to service the flush request. To prevent such deadlocks, our PRS is designed not to allocate memory dynamically during its life time. All memory used is allocated and locked at start-up time. Also, the PRS does raw I/O using the Linux `/dev/raw/rawN` interface, bypassing the buffer cache. Two additional changes are made in the kernel path used by the PRS to ensure that it never causes a PRD flush.

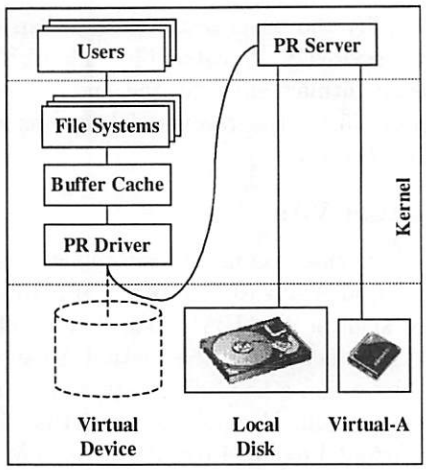


Figure 5: The PR Driver (PRD) and the PR Server (PRS) are the two main components of the PersonalRAID prototype. The PRD is implemented as a dynamically Loadable kernel module, whereas the PRS is a user-space process.

In addition to the in-memory map described in Section 3 (see Figure 4), the PRS maintains several main-memory segments for both the local disk and the VA device. These segments accumulate new block writes, just like the LLD system. Our implementation maintains more than one main-memory segment for each of the two devices. This feature allows us to decouple the amount of write-behind buffering from the segment size, which may need to be based on other considerations such as segment-cleaning performance.

Segment cleaning is begun when the number of clean segments falls below a threshold. We use twice the number of main-memory segments as the threshold. Cleaning is invoked, if necessary, after flushes and stops once the threshold is reached. For simplicity, on the local device, like the LLD system, it chooses the segment with the minimum number of live blocks to clean. Live blocks are read into main memory and are copied with their old timestamps to the main-memory segments. Note that we mix the

new block writes with the old blocks.

Segment cleaning for the VA device is a little more complex since replaying is integrated with cleaning. Our current cleaning policy adopts a simple heuristic to strike a balance between two goals: quickly generating empty segments on the VA and quickly propagating data. The cleaner gives higher preference to segments that have data blocks that need to be propagated. Among all such segments, a segment with the minimum number of such blocks is chosen. Among segments with no blocks that need to be propagated, we give preference to segments with fewer live blocks. To clean a VA device segment, live blocks are read into memory and if necessary, propagated to local disk. If the block still needs to be propagated to some other device, it is written back to the VA; otherwise, it is discarded. There is one subtlety in this algorithm. If a VA segment, while being cleaned, contributes new blocks to some main-memory segments of the local disk, then this VA segment cannot be reused until all those local disk segments are safely on the local disk. Otherwise, a crash may lead to a situation where the only copy of a data block is on a remote host, and recovering from such a crash would require visiting that remote host. Thus, before marking a set of segments cleaned in the current cleaning phase as free, we check for this condition and possibly flush the main-memory segments of the local disk.

Currently, the PRS performs request satisfaction, segment flushing, and cleaning sequentially. It is possible that performance can be improved by using multiple threads for some or each of these different tasks. The PRS is about 4000 lines of C code, including a substantial amount of debugging and testing code. The PRD is implemented as a dynamically loadable kernel module. It is about 700 lines of C code. All of the algorithms described above have been implemented with the exception of crash recovery, recovery from device loss, and reconfiguration.

5 Experimental Results

In this section, we evaluate the performance of the prototype PersonalRAID system to demonstrate two conclusions: (1) PersonalRAID can achieve the transparency and reliability goals without imposing a significant performance penalty on a mobile storage user, and (2) the log-structured organization of the PersonalRAID is a sound design choice.

5.1 Experimental Platform

Our experimental setup consists of two laptop end hosts (A & B) with an IBM 1GB Microdrive acting as the VA device. Table 1 shows the configuration of the laptops and Table 2 gives the characteristics of the laptops' internal local disks and the Microdrive. Note that the Microdrive's bandwidth is more than an order of magnitude worse than that of the internal disk. A challenge to the PersonalRAID system is to shield the user from this performance gap.

Model	Dell Inspiron 4000 Notebook
Processor	Pentium III, 800 MHz
Memory	256 MB
Operating System	Red Hat Linux 6.2 Kernel 2.2.18

Table 1: Host configuration.

	Local Disk	VA
Maker	IBM	IBM
Model	Travelstar 20GN	1 GB Microdrive
Interface	ATA-4	PCMCIA
Capacity (GB)	10	1
RPM	4200	3600
Bandwidth (MB/s)	16.9	1.5
Avg. Latency (ms)	7.1	20.3

Table 2: Characteristics of the local disks and the VA used in the PersonalRAID prototype. The published sustained data rate of the Microdrive is 2.6 MB/s; while our best effort micro-benchmark on the drive yields 1.5 MB/s.

Table 3 shows the configuration parameters of the PR Server. The user "sees" a 2 GB PersonalRAID logical disk, which is larger than the VA capacity. The segment sizes and the numbers of outstanding segments are chosen so that the write-behind buffer of the local partition is smaller than that of the VA: the former is kept small so that the system limits the amount of data loss in case of a crash, while the latter is kept large to allow overwritten data to be deleted before it reaches the Microdrive and mask the higher latency. We must, however, remark that in our experiments, the large size of the VA buffers did not make a significant difference. The reason for this is that a Linux ext2 file system internally uses large buffers to absorb most of the overwrites. Thus, very few overwrites occur in the PersonalRAID buffers, which are at the logical disk level below the file system. In a system where the file system does less buffering, the impact of having large VA buffers might be significant.

5.2 Benchmarks

We report results for two benchmarks. The first is an enhanced version of the "Modified Andrew

	Local Partition	VA
Block Size (KB)	4	4
Seg. Size (MB)	0.5	1
Outstanding Segs	2	8
Size (GB)	3	1

Table 3: Configuration parameters of our 2GB PersonalRAID server.

Benchmark" [4, 9], which we call "MMAB". (We modified the benchmark because the 1990 benchmark does not generate much I/O activity by today's standards.) MMAB has four phases. The first phase creates a directory tree of 50,000 directories, in which every non-leaf directory (with the exception of one) has ten subdirectories. The second phase creates one large file and many small files. The large file created has a size of 256 MB. Each of the small files is 4 KB. The benchmark creates five small files in each of the directories in the first five levels of the directory tree, resulting in a total of about 55,000 small files. The third phase performs file-attribute operations. During this phase, the benchmark first performs a recursive `touch` on all the directories and files in the directory tree; it then computes disk usage of the directory tree by invoking `du`. The fourth and final phase reads files. It first performs a `grep` on each file; it then reads all the files again by performing a `wc` on each file. We run the MMAB benchmark on laptop A and evaluate various options of gaining access to the resulting files on laptop B without the benefit of a network.

The second benchmark is a software development workload. We examine the cost of installing and compiling the Mozilla source code as a user moves between the two laptops. The benchmark has two phases. The first phase creates a development source tree from a compressed archive file (a `.tar.gz` file) stored on a local disk. The source tree consumes about 405 MB of total disk space. We refer to this phase as "MOZ1" in the subsequent sections. During the second phase, we compile the "layout" module within the Mozilla source tree, generating an additional 80 MB of data. We call this phase "MOZ2". To run this benchmark, we start with MOZ1 on laptop A. We then transport the data to the disconnected laptop B to continue with MOZ2.

5.3 Recording Performance

Table 4 details the recording performance of the MMAB benchmark. "UFS-Local" is a Linux "ext2" file system created directly on the local disk partition. "UFS-Upcalls" is the same file system implemented with kernel upcalls into a user-level server. "LLD-Local" uses the same kernel upcall and user-level server mechanisms but it replaces the UFS with a log-structured logical disk organization. The results of these experiments are used to establish the base-line performance, to factor out the cost of using a user-level server, and to quantify the benefit of using an LLD for accessing the local disk. "UFS-MD" is an ext2 file system created directly on the Microdrive. "PR-VA" is an ext2 file system created on the logical disk exported by the PersonalRAID system where the Microdrive is used as the VA device. "PR-VVA" is similar to PR-VA except that it uses a partition on the local disk as the VA device. The PR-VVA performance is an indication of how well the PersonalRAID might perform if the VA device has much better performance than that of the Microdrive.

The Linux ext2 file system performs *both* metadata and data writes asynchronously to the buffer cache. The large memory filters out overwritten data before it reaches the disks and allows the surviving write requests to be intelligently scheduled. As a result, the additional benefit that the log-structured PersonalRAID derives from asynchronous writes is smaller than one might expect. During the MMAB and MOZ experiments, the segment cleaners in the PRS did not get invoked. Later in this section, we describe a separate experiment designed to measure the overhead of segment cleaning.

The results show that the PR-VA system is successful in masking the 10× bandwidth difference between the local disk and the Microdrive: despite the fact that the PersonalRAID needs to write to two devices and that the system incurs the cost of kernel upcalls, the performance of PR-VA is close to or better than that of UFS-Local in most cases due to the relatively low overhead of log-structured recording. An exception is the large write (lwrite) performance, a case where the PersonalRAID recording performance is being limited by the Microdrive bandwidth. The read performance of the PR-VA is also excellent, since unlike the UFS-MD system which performs reads from the slower Microdrive, it satisfies reads from the faster local partition. Finally, the performance of PR-VVA indicates that the PersonalRAID system becomes even more attractive with a faster mobile storage technology.

Table 5 presents the recording performance of the MOZ benchmark, along with the cumulative totals from MMAB. Recall that MOZ1 is the source-unpacking phase run on laptop A and MOZ2 is the compiling phase run on laptop B. The former is more I/O intensive than the latter and it is much more difficult to mask the recording overhead during MOZ1 unless the VA device is faster. After we connect the

	mkdir (s)	lwrite (s)	swrite (s)	touch (s)	du (s)	grep (s)	wc (s)	total (s)
UFS-Local	156	20	398	806	6	142	368	1896
UFS-Upcalls	397	33	675	951	15	220	388	2679
LLD-Local	184	28	229	330	5	66	94	936
UFS-MD	398	320	725	1440	1200	535	880	5498
PR-VA	325	239	484	350	167	85	127	1777
PR-VVA	192	55	263	336	151	75	100	1172

Table 4: Detailed breakdown of recording performance for the MMAB benchmark. `mkdir` *is the directory creation phase.* `lwrite` *creates a large file and* `swrite` *creates many small files.* `touch` *and* `du` *perform attribute operations. The* `touch` *phase reads directories and inodes, and writes inodes. The* `du` *phase generates write as well as read traffic because the recursive visit alters access times that are stored in the inodes. The metadata cache misses are the main contributor of this phase's latency.* `grep` *and* `wc` *read all the files.*

	MMAB total (s)	MOZ1 (s)	MOZ2 (s) post-replay	MOZ2 (s) pre-replay
UFS-Local	1896	75	476	—
UFS-Upcalls	2679	126	481	—
LLD-Local	936	66	464	—
UFS-MD	5498	404	640	—
PR-VA	1777	395	548	558
PR-VVA	1172	102	482	486

Table 5: Recording performance.

	Cleaner not invoked (s)	Cleaner invoked (s)
PR-VA	878	928
PR-VVA	713	735

Table 6: Effect of cleaning on recording performance.

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
PR-VA	7.3	7.3	7.3
`tar`	1225	268	—
`hoard`	—	—	62

Table 7: Disconnection performance.

VA device to laptop B, we have two options for PR-VA: we can run MOZ2 immediately after connection, in which case the system reads data from the slower Microdrive; or we can run MOZ2 after replaying the entire content of VA to the local partition, in which case the system reads data from the faster local disk. The workload being CPU-intensive, there is little difference between the 2 cases.

The next set of experiments (see Table 6) capture the effect of invoking the segment cleaner on recording performance. We repeatedly perform a series of recompilation steps where each compilation step is triggered by modifying the file attributes of a small randomly-chosen subset of source files. We perform the experiments on two configurations: a 1 GB VA partition and a 550 MB VA partition. The disk utilization of the first configuration is 50% and the cleaner is not triggered during compilation. The disk utilization of the second configuration is 96% and the cleaner must run to continuously generate clean segments to accommodate the new data generated by the compilation. A total of 231 MB of data is generated during the experiment. The cost attributed to the cleaner is low.

5.4 Disconnection and Connection Performance

Table 7 and 8 compare the disconnection and connection latencies of several alternatives. To disconnect/connect the VA device from/to a host in PersonalRAID, all it takes is writing/reading the VA checkpoint. The checkpoint write may be preceded by the flushing of the remaining memory segments, which may add a few more seconds. The benchmarks that we used sync the disk at the end of each benchmark run so both the checkpoint write and read times are constants.

We examine two simpler alternatives to PersonalRAID. One is to use the Unix `tar` utility to create/unpack a Unix archive on the Microdrive at disconnection/connection times. `tar` writes less data to the Microdrive than PersonalRAID because it writes only enough information to allow it to recreate the directory structure without physically copying all the blocks, eliminating fragmentation costs. Unpacking time is much faster than packing time because unpacking is benefitting from the asynchronous writes of the Linux ext2 file system. Despite these optimizations, the latencies are not tolerable.

For MOZ2, `tar` alone would not have been adequate because we need to identify the files that have been changed, added, or deleted after compilation and act on just the changes. For this purpose, we use a pair of Unix scripts `hoard/unhoard`. Although these scripts are admittedly crude due to the liberal forking of some Unix processes, it is clear from the data that the resulting latencies of this approach are

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
PR-VA	7.0	7.0	7.0
`untar`	411	225	—
`unhoard`	—	—	67

Table 8: Connection performance.

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
VA→LFS	553	340	56
VA→UFS	988	547	100
VVA→LFS	159	90	17

Table 9: Replaying performance.

unlikely to be satisfactory.

To be fair, we note that the connection times for the simpler alternatives described above include the replay time. These times are less than the sum of connection and replay times for the PersonalRAID system (see the top rows in Tables 8 and 9), because the simpler alternatives write less data to the Microdrive than PersonalRAID, and write their data as a large, sequential file. Unlike PersonalRAID, however, these simpler alternatives do not allow normal operations to overlap the replay time and thus have a much greater impact on the user.

5.5 Replaying Performance

After a VA device is connected to a PersonalRAID host, background propagation (or replaying) starts. Table 9 shows the results of the experiments designed to analyze the impact of the log-structured organization on the replaying performance. "VA→LFS" refers to the PersonalRAID that replays from a log-structured VA to a log-structured local partition.

An alternative to this design is to use a UFS-style update-in-place organization on the local disk partition, while retaining the log-structured organization on the VA so that reads for replaying still occur at segment-sized granularity. To realize this alternative design, all we need to change is the L-to-P mapping for the local disk; we substitute it with an "identity mapping", which simply maps a logical address received from the file system to an identical physical address. The PR Server employs helper threads to perform asynchronous writes to the local partition and it limits the maximum number of outstanding writes to 20. Table 9 refers to this alternative as "VA→UFS".

Finally, to evaluate the impact of a faster mobile storage device, we replay from a local partition-based VVA (as described in Section 3.7) to another local partition, both of which are log-structured. Table 9 refers to this last alternative as "VVA→LFS".

During these experiments, the cleaner on the log-structured local partition did not get invoked. Both cleaning and replaying are background activities that can result in synergistic benefits. We plan to investigate the performance impact of this integration in the near future. Furthermore, since all the storage devices in a PersonalRAID are self-contained, it is possible for end hosts to independently clean their local disks when they are not in use. Therefore, it is possible that large number of free segments are available before replaying starts.

Table 9 shows that despite the fact that both the VA→LFS and the VA→UFS configurations are limited by the slow read performance of the Microdrive, the former can replay significantly faster than the latter, thanks in no small part to the former's log-structured organization. If the VA device could perform reads faster, the impact of the log-structured organization would have been even more dramatic as implied by the VVA→LFS performance numbers.

6 Related Work

Although PersonalRAID can be extended to deal with conflicting updates (as discussed in Section 3.8.1) and this is one of our ongoing research topics, the primary use cases targeted by PersonalRAID today are just that: personal usage scenarios where the availability of a single coherent name space and reliability are the primary concerns but conflict resolution is not. Conflicts are inherently file system-level or application-level events that must be addressed at these higher levels. PersonalRAID is a storage-level solution that can only provide mechanisms such as versioning that higher level systems may exploit. A number of research systems (including Ficus [11], Coda [6], and Bayou [10, 15]) have focused on conflict resolution techniques that may provide insight on how to build and extend services running on top of PersonalRAID.

Many systems share PersonalRAID's goal of synchronizing the contents of a number of hosts; these systems include disconnected client/server systems such as Coda [6], distributed applications such as those built on top of Bayou [10, 15], and replicated databases such as those provided by Oracle [8] and Sybase [2]. A common technique is to use an operations log that is recorded at the site that initiates the updates and is replayed at the various replicas. A variation of the theme is the use of "asynchronous RPCs" as those employed by Rover [5]. Before log replaying is complete, the access to a replica needs to be suspended if one does not want to expose stale data. We have determined that neither stale data

nor the latency involved in log propagation may be tolerable for a PersonalRAID user. Thanks to its LFS roots, PersonalRAID inherits the absence of the notion of a separate operations log—the collection of distributed logs *is* the storage system. As a result, the fresh updates carried in the VA device are always immediately accessible while replaying can occur in the background when convenient.

A VA device is similar to a disconnected Coda client in that it "hoards" data and buffers the latest changes [6]. It is different in that the VA device is not meant to support I/O operations on its own: the role of the VA in a PersonalRAID is two fold: (1) it acts as a transporter of all updates that are used to synchronize the contents of several disconnected end hosts, some of which may be mobile; and (2) it supports I/O operations only when it is coupled with a host local disk. There is no user involvement or guesswork involved in determining the content of the VA; and there is no danger of a "hoard miss."

The use of a mobile device to carry updates to other weakly connected hosts to bring about eventual consistency via pair-wise communications is similar to the approach taken by Bayou [10, 15]. Bayou provides a framework for application-specific conflict resolution and applications must be reprogrammed or developed from scratch to take advantage of the Bayou infrastructure. PersonalRAID, as a storage system, does not resolve conflicts in itself; as a result, it is possible for us to develop a general system, on top of which existing personal applications may run unmodified.

Existing mobile systems typically do not address data reliability on mobile hosts: these mobile hosts are typically considered inferior "second-class citizens" and their data is vulnerable until they are propagated to "first-class citizen" servers that are professionally managed and backed up occasionally. PersonalRAID provides protection against any single device loss at all times by leveraging the exact same mechanism that is needed to bring about eventual consistency of the system.

Finally, there are existing applications such as the Windows "Briefcase" that can synchronize the contents of multiple hosts. The two problems with these applications are exactly the problems that PersonalRAID is designed to address: (1) the inconvenience involved in manual movement of data, and (2) the poor performance in terms of both latency and throughput during synchronization events.

7 Conclusion

As storage technology advances, a user is facing an increasing array of disconnected storage devices. Two of the important challenges a user must face is the lack of a single transparent storage space that is ubiquitously available and a certain degree of reliability assurance. PersonalRAID is a mobile storage management system that attacks these two problems. At the heart of a PersonalRAID is a mobile Virtual-A device that allows the user to transparently transport, replicate, and access data while interacting with a number of disconnected storage devices. By employing a distributed log-structured organization, the system is able to accomplish these tasks without imposing any serious performance penalty on the user.

Acknowledgments

We would like to thank Brent Waters, Victor Shnayder and Atul Pokharel for their ideas and help during the initial part of the project. Thanks are also due to our shepherd David Kotz and other FAST reviewers for their detailed and critical comments.

References

[1] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[2] GORELIK, A., WANG, Y., AND DEPPE, M. Sybase Replication Server. In *Proc. ACM SIGMOD Conference* (May 1994), p. 468.

[3] GROWCHOWSKI, E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In *Datatech* (September 1998), ICG Publishing, pp. 11–16.

[4] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[5] JOSEPH, A. D., DELESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., AND KAASHOEK, M. F. Rover: A Toolkit for Mobile Information Access. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 156–171.

[6] KISTLER, J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[7] LUMB, C., SCHINDLER, J., GANGER, G. R., RIEDEL, E., AND NAGLE, D. F. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[8] ORACLE CORPORATION. *Oracle7 Server Distributed Systems: Replicated Data*, 1994.

[9] OUSTERHOUT, J. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. of the 1990 Summer USENIX* (June 1990).

[10] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 288–301.

[11] REIHER, P., HEIDEMANN, J., RATNER, D., SKINNER, G., AND POPEK, G. Resolving File Conflicts in the Ficus File System. In *Proc. the Summer USENIX Conference* (June 1994), pp. 183–195.

[12] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[13] SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[14] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[15] TERRY, D. B., THEIMER, M. M., PETERSON, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 172–183.

Hippodrome: running circles around storage administration

Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, Alistair Veitch
Storage and Content Distribution Department
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304
{anderse,mjhobbs,kkeeton,suspence,uysal,aveitch}@hpl.hp.com

Abstract

Storage system configuration, even at the enterprise scale, is traditionally undertaken by human experts using a time-consuming process of trial and error, guided by simple rules of thumb. Due to the complexity of the design process and lack of workload information, the resulting systems often cost significantly more than necessary, or fail to perform adequately.

Our solution to this problem is to automate the design and configuration process using a tool we call Hippodrome. It can explore the design space more thoroughly than humans, and implement the design automatically, thereby eliminating many tedious, error-prone operations.

Hippodrome is structured as an iterative loop: it analyzes a workload to determine its requirements, creates a new storage system design to better meet these requirements, migrates the existing system to the new design. It repeats the loop until it finds a storage system design that satisfies the workload's I/O requirements. This paper describes the Hippodrome loop and demonstrates that our prototype implementation converges rapidly to appropriate system designs.

1 Introduction

Enterprise-scale storage systems are extremely difficult to manage. The size of these systems, the thousands of configuration choices, and the lack of information about workload behaviors raise numerous management challenges. Users' demand for larger data capacities, more predictable performance, and faster deployment of new applications and services exacerbate the management problems. Worse, administrators skilled in designing, implementing and managing these storage systems are expensive and in short supply. It is estimated that the cost of managing storage is several times the purchase price of the storage hardware [1, 25]. These difficulties are beginning to cause enterprise customers to out-source their storage needs to Internet data centers and storage service providers, such as Exodus [18], who will lease networked storage. The growing importance of this storage model implies that the ability to accurately provision storage systems to meet workload needs will become even more critical in the future.

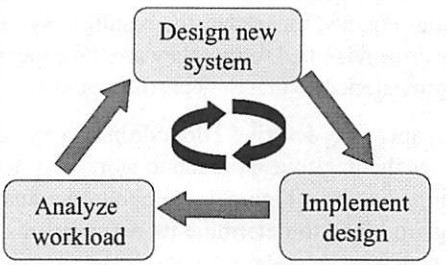


Figure 1: The stages of the iterative storage management loop. The loop can be bootstrapped using capacity information and optional performance estimates as input to the design stage.

Storage management challenges include designing and implementing the storage system, adapting to changes in workloads and device status, designing the storage area network [32], and backing up the data [27]. In this paper, we concentrate on the important problem of *storage system configuration*: designing and implementing the storage system needed to support a particular workload, before the storage system is put into production use.

Given a pool of storage resources and a workload, we want to determine how to automatically choose storage devices, determine the appropriate device configurations, and assign the workload to the configured storage. These tasks are challenging because the large number of design choices may interact with each other in poorly understood ways. To make reasonable design choices, administrators need detailed knowledge of applications' storage behavior, which is difficult to obtain. Once a

design has been determined, implementing the chosen design is time-consuming, tedious and error-prone. A mistake in any of the implementation operations is difficult to identify, and can result in a failure to meet the performance requirements of the workload.

Storage system configuration is naturally an iterative process, traditionally undertaken by human experts using "rules of thumb" gained through years of experience. They start with a first design based on an initial understanding of the workload, and then successively refine the design based on the observed behavior of the system. Figure 1 illustrates this iterative loop. Unfortunately, the complexities of the systems being designed, coupled with inadequate information about the true workload requirements, mean that the resulting systems are often over-provisioned so that they are too expensive, or under-provisioned so that they perform poorly.

In this paper, we describe Hippodrome, a system that automates the iterative approach to storage system configuration shown in Figure 1. Hippodrome analyzes a running workload to determine its requirements, calculates a new storage system design, and migrates the existing system to the new design. Hippodrome makes better design decisions by systematically exploring the large space of possible designs. Hippodrome decreases the chance of human error by automating the configuration tasks. As a result, Hippodrome frees administrators to focus on the applications that use the storage system.

We show that Hippodrome generates storage system configurations that employ near minimal resources to satisfy workload requirements, and that it converges to the final system design in a small number of iterations.

The remainder of this paper is organized as follows. Section 2 describes the automation of storage system configuration, including its goals and challenges. Section 3 introduces our solution, Hippodrome, and its components. Section 4 describes our experimental methodology and presents our results on random-access workloads and the PostMark filesystem benchmark. Section 5 discusses related work, Section 6 summarizes our results, and Section 7 describes directions for future research.

2 Problem statement

The iterative approach to system management shown in Figure 1 is applicable to many levels of the system, including the block-level array subsystem, the filesystem and the application itself. We focus on the block-level storage, as it provides a potential benefit to all applications that store data, including those that use the filesystem and those that use the raw block interface directly.

We define the three stages of the iterative storage management loop as follows:

- **Design new system**: Design a system to match the current workload requirements. This stage includes choosing which storage devices to use, selecting their configurations, and determining how to map the workload's data onto the configured devices. The requirements may come from observations of the workload behavior in previous iterations.
- **Implement design**: Configure the disk arrays and other storage system components, enable access to the storage resources from the hosts, and migrate the existing application data (if any) to the new design.
- **Analyze workload**: Analyze the running system to learn the workload's behavior. This information can then be used as input to the design stage in the next iteration.

We want to remove the human administrators from the loop as much as possible, by automating the iterative loop, to the point where all that is required at the beginning is workload capacity information. The loop will then learn the performance requirements across multiple iterations of the loop.

In order to be considered successful, the automated loop must meet two goals. First, it must converge on a viable design that meets the workload's requirements without over- or under-provisioning. Second, it must converge to a stable final system as quickly as possible, with as little input as possible required from its users.

2.1 Definitions

A *workload* is the set of requests observed by the storage system. A particular workload may be generated by one or more applications using the storage system. We describe a workload in terms of *stores* and *streams*. A store is a logically contiguous chunk of storage. A stream captures information about the I/O accesses to a single associated store, such as average request rate and average request size. Section 3.1 describes the characteristics of streams. Expressing a workload in terms of stores and streams decouples the specification of the workload from the application(s) that generate that workload. As a result, our workload specification and assignment techniques are applicable to a broad range of applications.

Disk array storage is divided into *logical units (LUs)*, which are logically contiguous arrays of blocks exported by a disk array. LUs are usually constructed by binding

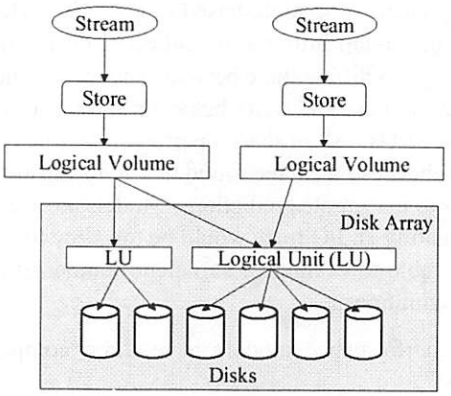


Figure 2: Storage and workload concepts. Stores and streams characterize the workload. LUs are containers for data. A store is implemented as a logical volume, which is used to map the store onto one or more LUs.

a subset of the array's disks together using RAID techniques. LU sizes are typically fixed by the array configuration, and so are unlikely to correspond to application requirements.

Logical volumes add flexibility by providing a level of virtualization that enables the server to split the (large) LUs into multiple pieces or to stripe data across multiple LUs. A logical volume provides the abstraction of a virtual disk for use by a filesystem or database table.

We implement a store as a logical volume in our system. Figure 2 illustrates the relationships between stores, streams, LUs and logical volumes.

3 Hippodrome

Hippodrome is our iterative design tool for storage systems. This section describes the components of the Hippodrome loop in more detail, and explains how the components interact to design a storage system iteratively.

The Hippodrome components are the most recent versions of our group's ongoing research into storage system modeling and configuration [2, 3, 4, 5, 6, 10, 26, 30, 31, 32]. The goal of this paper is to show how the different components can work together to automate storage management. Therefore, this section summarizes the techniques used in Hippodrome. The details on each component may be found in the paper on that topic.

Hippodrome uses four interdependent components to implement the iterative loop shown in Figure 1. The *analyze workload* stage summarizes a workload's behavior. This summary is used to predict the workload's requirements in the next iteration. Two components cooperate to implement the *design new system* stage: *performance models* for the storage devices, and a design engine, or *solver*. The performance models predict the utilization of storage devices under a candidate workload. The solver designs a new storage system using the performance models to guarantee that no device in the design is overloaded. The *implement design* stage migrates any existing system to the new design.

The different components share the responsibility for the correct operation of the loop. Collectively, they provide:

- **Accurate resource estimation.** The analysis and model components cooperate to accurately predict the utilizations of arbitrary candidate configurations. These utilizations are computed relative to the maximum performance capabilities of the devices, with utilizations at or above 100% indicating that the system is unable to support the desired performance of the workload, and utilizations under 100% indicating that the system can support the desired workload performance. The models and analysis encapsulate the physical performance characteristics of the devices and workloads, allowing the solver to concentrate on the optimization task.
- **Minimal designs.** The solver is responsible for choosing a design that uses the least resources among the set of candidate *valid designs*: ones that meet the performance requirements of the workload and ensure that all of the components in the design are under 100% utilized. It should perform this search quickly.
- **Balanced designs.** The solver is responsible for finding *balanced designs*: ones that have the minimal utilization variation across the storage system resources. Balanced designs allow the system to grow more quickly in each loop iteration, because they afford more opportunity for incorporation of new resources as the loop iterates.
- **Short migration time.** The migration component is responsible for converting the existing system into the new proposed system. It should perform this migration efficiently, without requiring significant temporary storage.

Accurate resource estimation and minimal designs result in correctly provisioned systems. Balanced designs and short migration time enable the loop to configure storage systems quickly.

We describe these components and their inputs/outputs in the sections below, focusing on how each component contributes to the operation of the Hippodrome loop.

3.1 Analysis component

The analysis component takes as input a detailed block-level trace of the workload's I/O references and a description of the storage system (LU and logical volume layouts). It outputs a summary of the trace in terms of stores and streams [31]. The analysis component captures enough properties of the I/O trace in the streams to enable the models to make accurate performance predictions.

The analysis component models an I/O stream as a series of alternating ON/OFF periods, where I/O requests are only generated during ON periods. More specifically, we define the minimum duration of an ON period, *minOnTime*, as 0.5 seconds, and the minimum duration of an OFF period, *minOffTime*, as at least two seconds of inactivity.

During an ON period, we measure six parameters for each stream: the mean read and write *request rates*; the mean read and write *request sizes*; the *run count*, which is the mean number of sequential requests; and the *queue length*, which is the mean number of outstanding I/O requests. Because streams can be ON or OFF at different times, we also capture inter-stream phasing and correlations using the *overlap fraction*, which is approximately the fraction of time that two streams' ON periods overlap. (The formal definition is slightly more involved and is described in [10].) Table 1 provides a summary of all of the stream attributes.

We choose to trace the I/O activity and analyze it later (or on another machine) to minimize the interference with the workload. Capturing I/O trace data results in a CPU overhead of 1-2% and an increase in I/O load of about 0.5%. Even day-long traces are typically only a few gigabytes long, which is a negligible storage overhead as the trace only has to be kept until the analysis is run. The duration of tracing activity is workload dependent, as it has to cover the full range of workload behavior. For simple workloads, a few minutes may be sufficient. For complex workloads, it may take a few hours.

3.2 Performance model component

The performance model takes as input a workload summary from the analysis component, and a candidate storage system design from the solver. The candidate design specifies both the parameters for the storage system and the layout of stores onto the storage system. It outputs the utilization of each component in the storage system.

The model component needs to predict storage system performance quickly and accurately. We implement this component using table-based models [3]. The models use the stream information collected during the analysis stage to differentiate between sequential and random behavior, read and write behavior and ON-OFF phasing of disk I/Os. All of the properties shown in Table 1 are used, because we have found that ignoring any of them leads to inaccurate predictions. Models are used because simulating an I/O trace would be too slow for the solver to be able to examine a sufficient number of candidate configurations.

The performance models have three complementary parts:

1. **Inter-stream adjustments.** The input queue length and sequentiality are first adjusted to take into account the effect of interactions between streams on the same LU using the techniques described in [30]. For example, the sequentiality is decreased for two streams that are on simultaneously, because the overlap will cause extra seeks, while the queue length is increased because there will be more outstanding I/Os, which gives the disk array more opportunity for request re-ordering to improve performance.

2. **Single-stream prediction.** The utilization of each stream is calculated using a table of measurements [3]. The model looks up the nearest table entries to the specified input values for the stream, and then performs a linear interpolation to determine the maximum request rate at those values. The utilization is the mean request rate of the stream divided by the maximum request rate.

3. **Utilization combination.** The model calculates the utilization of each LU by combining the estimated stream utilizations using the phasing algorithms found in [10]. The algorithms ensure that the utilization of two streams is proportional to the fraction of time that they overlap.

3.3 Solver component

The solver [5] reads as input the workload description generated by the analysis component, and outputs the design of a system that meets the workload's performance requirements. The output specifies a number of disk arrays, the configuration of those arrays (e.g., number of disks, LU configurations, controller and cache settings) and a mapping of the stores in the workload onto the disk arrays' LUs.

Attribute	Description	Units
request_rate	mean rate at which requests arrive at the device	requests/sec
request_size	mean length of a request	bytes
run_count	mean number of requests made to contiguous addresses	requests
queue_length	mean size of the device queue	requests
on_time	mean period when a stream is actively generating I/Os	sec
off_time	mean period when a stream is not active	sec
overlap_fraction	fraction of the "on" period when two streams are active simultaneously	fraction

Table 1: Workload characteristics generated by Hippodrome's analysis stage, and used by its models.

The solver efficiently searches the exponentially large space of storage system designs to find a balanced, valid, minimal design. The problem of efficiently packing a number of stores, with both capacity and performance requirements, onto disk arrays is similar to the problem of multi-dimensional bin packing. Since bin-packing is an NP-complete problem, exhaustive searches would take too long. Therefore our solver builds on the best-fit approaches found in [15, 21, 23] to produce initial solutions, and adds backtracking to help the solver avoid local minima in the search space of possible designs.

The solver algorithm has three phases:

1. **Initial assignment.** This phase attempts to find an initial, valid solution. It first randomizes the list of input stores, and then individually assigns them onto a growable set of LUs. It assigns each store onto the best available LU. Because the goal is to minimize the cost of the system, the best available LU is the one that is closest to being full after the addition of the store. If the store does not fit onto any available LU because the resulting utilization or capacity would be over 100%, the solver expands the storage system.

2. **LU re-assignment.** This phase attempts to improve on the solution found in the first phase. The solver uses randomized backtracking to avoid the local minima that can result from the first phase. It then randomly selects an LU from the current design, removes all the stores from it, and re-assigns those stores in a similar manner to the assignments done in the first phase. This operation is repeated until all of the LUs have been reassigned. At the end of this phase, we have a near-optimal but potentially unbalanced assignment of stores to LUs, using the minimum necessary storage resources.

3. **Store re-assignment.** This phase load-balances the best solution found in phase two. The load is measured as the utilizations of the components (e.g., LUs, disk-array controllers) predicted by the models. The solver repeatedly selects a store at random, removes it from the assignment and then re-assigns it, but in this phase with the goal of producing a more balanced solution. The solver has already packed the stores tightly in the first two phases, and guarantees that the balanced solution does not increase in cost.

Experiments with this solver have found that it produces near optimal solutions. The optimal solution is a balanced valid design that meets the workload requirements with minimal set of resources. For most cases where we can prove optimality, the solver generates optimal solutions. We have also compared the solver to an exhaustive search algorithm on small cases, and again found that the solver finds optimal solutions. We have hand designed some pessimistic inputs (since the problem is NP-complete, these must exist), and found that the solver generates solutions about 10% worse than optimal on those inputs. In comparisons with other solver algorithms [2], we have found that our solver generates solutions that are as good or better. More details on the solver can be found in [5].

3.4 Migration component

The migration component takes as input the new design of the storage system, and changes the existing configuration to the new design. It configures storage devices, copies the data between old and new locations, and changes the parameters of the storage system to match the parameters in the new design.

Migration operates in two phases. First a plan is generated for the migration and then the plan is executed. The planning phase tries to minimize the amount of scratch space used and the amount of data that needs to be moved. The problem of migration planning for variable-sized objects is NP-complete, as it is reducible to subset sum [17]. We use a simple greedy heuristic that moves stores to their final location. If no store can be moved to its final location in a single step, the heuristic

chooses a candidate store (or set of stores, if the underlying device needs to be reconfigured) and moves all of the stores blocking the move of the selected store into scratch space. The heuristic selects the candidate store to minimize the amount of scratch space needed. The result is a sequential plan for the migration.

If the underlying logical volume manager allows individual logical blocks to be moved, as opposed to an entire volume (store), then more advanced algorithms [4] that generate efficient parallel plans can be used.

Second, in the execution phase, the migration component copies the stores to their destinations as specified by the plan. The migration can be executed with the workloads either online or offline. Offline migration creates a new logical volume, copies the data there, and deletes the original volume. Online migration allows the workloads to continue executing. It uses the LVM to mirror the volume to its new location, and then splits the mirror, removing the old half. The techniques in [26] can be used to minimize the performance impact on the workload.

An alternative method that works during initial system configuration involves configuring the devices and then copying the data from a "master copy" of the stores to their final destinations. This approach works well if the design is changing substantially between iterations, but requires double the storage capacity to hold the master copy.

3.5 Putting it all together

We now consider how the Hippodrome components work together to find a storage system that supports the user's target workload so that the storage is not the bottleneck resource (i.e., the predicted utilization of all components is less than 100%). Since the performance of this target is unknown, Hippodrome iteratively estimates the target workload requirements by repeatedly generating and implementing storage designs. The estimation is performed by running the workload against the resulting system and monitoring and analyzing the I/O behaviors (as described in Section 3.1) to develop a new estimate of the workload requirements.

At each iteration of the loop, Hippodrome uses the new workload estimate to develop a storage system design to accommodate it. This design is created from scratch based only on the current estimate of the workload's requirements.

In searching the space of possible configurations, the solver will evaluate configurations with an increasing amount of resources. If it determines that all designs with the same amount of resources as in the current design would be over-committed, then we know that the storage system was the bottleneck, and the solver will find a design with more resources. If it determines that a design with fewer resources is sufficient, then we know that the storage system was under-utilized. Otherwise, it will find a configuration with the same amount of resources, and so we know that the loop has converged.

If the workload estimate was low because the storage system was over-committed, the newly designed system will contain more resources. The additional resources will allow the application(s) to increase their I/O performance, and hence increase the workload estimate. If the workload estimate is still too low, the process will repeat, until the workload appears to need no more resources.

For example, suppose the first iteration produces a design using 10 disk drives, based on only capacity information. Measurements from the first iteration might show the workload achieving 1000 I/Os per second (IOPS) on this configuration (because the bottleneck is the disk drives, which can perform 100 IOPS each). The second Hippodrome iteration might produce a design that incorporates 12 disk drives, because the Hippodrome models conservatively assume that the disk drives can only achieve 90 IOPS, and so 12 disk drives are required to support 1000 IOPS. With this configuration, the workload might then achieve 1200 IOPS, leading to a design with 14 disk drives. Finally, when running on the system with 14 disk drives, the workload might still run at 1200 IOPS, because the storage system is no longer the bottleneck. At this point Hippodrome has *converged*, and no longer increases the available resources.

The time to converge is determined by how many loop iterations must be performed and how long each iteration takes. The number of loop iterations depends on the size of the final system and the degree of mismatch between the initial design and the final design necessary to satisfy the workload's target performance requirements. Although Hippodrome performs well starting only with capacity requirements, Section 4 shows that Hippodrome can use an initial performance estimate to converge faster. The time for each iteration is dominated by running the application and implementing the design. Application run times can range from minutes to hours. Implementing the design can also take minutes to hours, because it involves moving some fraction of the (potentially sizeable) data in the system. Conversely, analyzing the workload and generating the new design takes seconds to minutes.

The number of iterations required may be influenced by the explicit addition of *headroom* (resource slack) in each step. For example, Hippodrome's user can ask it to keep the utilization level below 85%, rather than 100%.

This guideline will increase the likelihood that there will be resource slack in the resulting system, thereby giving the workload a greater chance to express itself. This opportunity may result in increased application I/O performance and hence an increased workload estimate after the current iteration. In turn, this can reduce the number of iterations needed to get to the target workload. Explicit headroom can be used to compensate for optimistic errors in the performance models, where the models predict that available performance is higher than the actual performance (resulting in an under-estimate of the resources needed to support a workload). Headroom can also be implicitly added. If the models predict that the available performance is lower than the actual performance (i.e., they are pessimistic), the resulting designs implicitly include resource slack. In addition, solver designs that balance the load across the storage devices provide the maximum room for growth (over imbalanced designs), as no single part of the system will be nearer to its utilization limits than any other.

Once the load has stabilized and the configuration converged, retaining headroom may be of lesser value. However, it can still be used to accommodate short-term variations in the workload and to provide for future growth. Ultimately, providing headroom is part of a risk-cost decision: reducing the risk of a mis-configured system comes at the cost of additional resources. In what follows, we set the headroom to zero, because we wish to evaluate Hippodrome in the most stringent conditions, without any such resource slack.

We now turn to an evaluation of the Hippodrome system.

4 Experimental results

In this section we describe the experiments we ran to evaluate Hippodrome. Our experiments are designed to answer the following questions:

- Does Hippodrome converge? If so, how fast?
- Does Hippodrome allocate a reasonable amount of resources for a given workload?

4.1 Experimental workloads

Our evaluation is based on three variants of a simple fixed-size, random-access workload and a modified version of the PostMark benchmark [22]. The random-access workloads are useful for validating whether the Hippodrome loop performs correctly, because we can determine the expected behavior of the system. The PostMark benchmark is useful because it lets us investigate how Hippodrome performs under a more realistic workload.

Parameter	Always on	Phased
Store size (MB)	1024	1024
Number of stores	100	100
Request size (KB)	32	32
Request rate (IOPS/stream)	12.5, 25	50
Request type	read	read
Request offset	1KB aligned	1KB aligned
Run count	1 (random)	1
ON/OFF periods (sec)	always on	4.5 / 5.5
Correlated Groups	n/a	2 groups
Arrival process	4-bounded Poisson	4-bounded Poisson

Table 2: Common parameters for each stream in the random-access workloads.

In our experiments, we use the workloads shown in Table 2 with fixed-size, random requests; generating a load that ranges from 12.5 to 50 I/Os per second (IOPS) for each individual stream. We also use workloads that exhibit complex phasing behavior where groups of streams have correlated ON/OFF periods. We generate these workloads using a synthetic load generator capable of controlling the access patterns of individual streams. For each stream, it generates an access pattern from the given request rate, request size, sequentiality (specified by the run count), maximum number of outstanding requests and the duration of ON/OFF periods. We used a modified Poisson arrival process in the random-access workloads that restricted each stream to having no more than 4 requests outstanding at a given time.

We also use the PostMark benchmark, which simulates an email system, in our experiments. The benchmark consists of a series of transactions, each of which performs a file deletion or creation, together with a read or write. Operations and files are randomly chosen. Using the default parameters, the benchmark fits entirely in the array cache, and exhibits very simple workload behaviors, so we have scaled the benchmark to use 40 sets of 10,000 files, ranging in size from 512B to 200KB. This provides both a large range of I/O sizes and sequentiality behavior. In order to vary the intensity of the workload, we run multiple identical copies of the benchmark simultaneously on the same filesystem. The data for the entire PostMark benchmark has been sized to fit within a single 50 GB filesystem. Hippodrome treats the filesystem as a single store, accessed by a single stream.

For each workload, we let Hippodrome generate an initial system design based solely on the capacity requirements and then iteratively improve the system design un-

til it converges to support the workload. We do not expect the loop to converge in a single step, because the workload may not be able to run at full speed on the initial capacity-only design. However, we show that the loop converges quickly and that providing initial performance estimates can speed up the convergence.

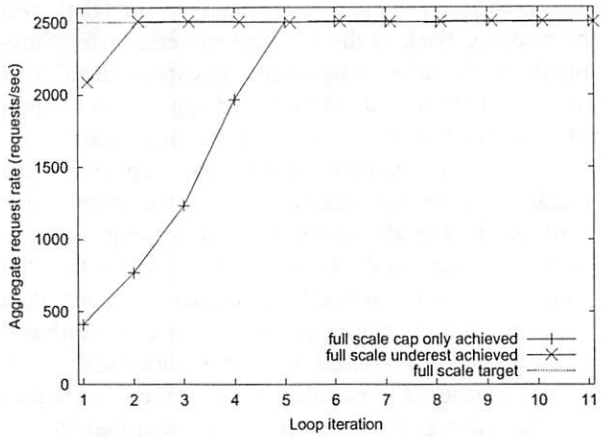
The migration step for the random-access workloads simply re-creates the logical volumes in the new locations. Those workloads accept arbitrary data in the logical volumes. The migration step for the PostMark workload copies the data from a master copy. Migrating the data would require reading and writing it from the same array since there is only a single store. Copying the data from the master speeds up our experiments because the array used for the experiments is only writing data during the migration.

4.2 Experimental infrastructure

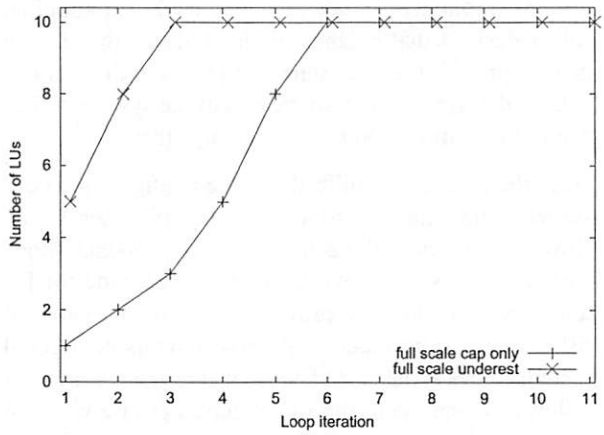
Our experimental infrastructure consists of an HP FC-60 disk array [20] and an HP 9000-N4000 server. The FC-60 array has sixty 36 GB Seagate ST136403LC disks, spread evenly across six disk enclosures. The FC-60 has two controllers in the same controller enclosure with one 40 MB/sec Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. Each controller can access all of the SCSI buses, and has 512 MB of battery-backed cache (NVRAM). Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. Each controller of the FC-60 is connected to a Brocade Silkworm 2800 switch via a 1 Gb/sec FibreChannel link. A particular LU can only be efficiently accessed through a single controller at a time, although each controller can access all of the LUs.

Our HP 9000-N4000 server has eight 440 MHz PA-RISC 8500 processors and 16 GB of main memory, and runs HP-UX 11.0. It uses a separate FibreChannel interface to access each of the controllers in the disk array.

We configured each of the LUs in the system as a six disk RAID-5 LU with a 16 KB stripe unit size. This configuration allowed us to avoid a multi-hour array reconfiguration time during each iteration, at the cost of restricting the solver to a subset of the possible array configurations. Although Hippodrome is capable of allocating physical resources in smaller units (e.g., different numbers of disks in an LU), and it already considers controller and bus resource limitations in its allocation, the restriction to fixed-size LUs is convenient for experimental purposes. The restricted design space also helped us determine whether Hippodrome had found the correct configuration, as it allowed us to analyze all the possible designs. We report the resources Hippodrome allocates in units of LUs; the reader can also think of this as "groups of six disks".



(a) Average request rate



(b) Number of LUs.

Figure 3: (a) Target and achieved average request rates at each iteration of the loop for the random-access workloads with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used during each iteration.

4.3 Random-access workloads

We start with fixed-size, random-access workloads so that it is easy to understand the behavior of the loop. We present two sets of results, one where all streams are ON at the same time (Section 4.3.1), and one where streams have correlated ON and OFF periods (Section 4.3.2). We compare our results to the target request rate to determine whether the system designed by Hippodrome has met the workload's requirements. However, this practice is for illustration only. Hippodrome has no knowledge of these target rates.

4.3.1 Always ON workloads

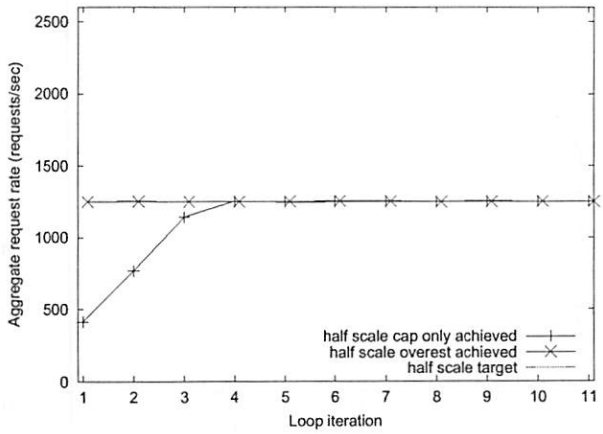
Figure 3(a) shows the target I/O rate and the achieved I/O rate for the random-access workloads at each iteration of the loop. The figure illustrates two sets of experiments with different input assumptions: one using only capacity information (labeled "cap only"), and one using an initial under-estimate of the performance (labeled "underest"). For the capacity-only design, we see that Hippodrome's storage system design converges within five loop iterations to achieve the target I/O rate of the workload (2500 requests per second). We also see that the initial guess cuts the convergence time down to two iterations.

Figure 3(b) shows the number of LUs allocated by Hippodrome at each loop iteration to achieve the target I/O rate. The system converges in five loop iterations starting from only capacity requirements. In the first four iterations, the LUs are over-utilized, and Hippodrome allocates new LUs, increasing the system size to better match the target request rate. As more LUs are added, a smaller fraction of the LUs' capacity is used for the workload's data. As a result, the seek distances got shorter and the disk positioning times are reduced. However, our performance models were calibrated using the entire disk surface, and therefore slightly under-estimate the performance of the LUs when a fraction of an LU is used. As a result, Hippodrome allocates two more LUs at the fifth iteration, even though the application could achieve its target rate without this.

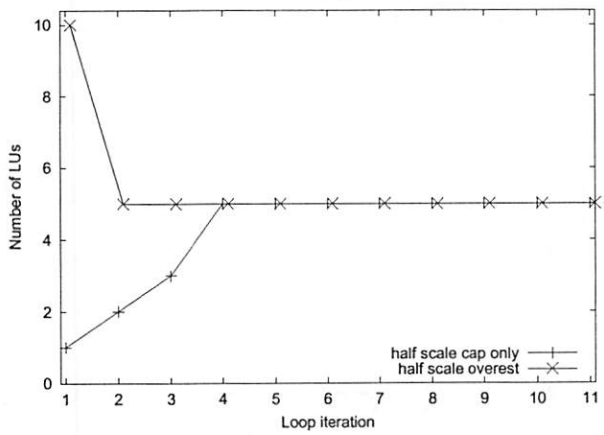
Iterations for the random-access workloads are extremely quick. We run the workload generator for 5 minutes. The analysis and solver take at most a few minutes. The implementation takes a few minutes to re-build the logical volumes, but as the synthetic generator does not have any data, we skip the step of copying data onto the logical volumes.

These results show that Hippodrome can rapidly converge to the correct system design, using only capacity information as its initial input.

Figure 4 shows that Hippodrome uses the minimal number of resources necessary to satisfy the workload's performance requirements. The target request rate for both workloads is 1250 requests per second, which can be achieved using only five LUs. Given only capacity requirements as a starting point, the loop converges to the target performance and correct size in three iterations. Given an initial (incorrect) performance estimate that the aggregate request rate is 2500 requests per second (twice the actual rate), the loop initially over-provisions the system to use 10 LUs, easily achieving the target performance. The analysis of the actual workload behavior in the first iteration produces workload requirements that Hippodrome can accommodate with fewer resources, and Hippodrome scales back the system to use five LUs in the second iteration.



(a) Average request rate

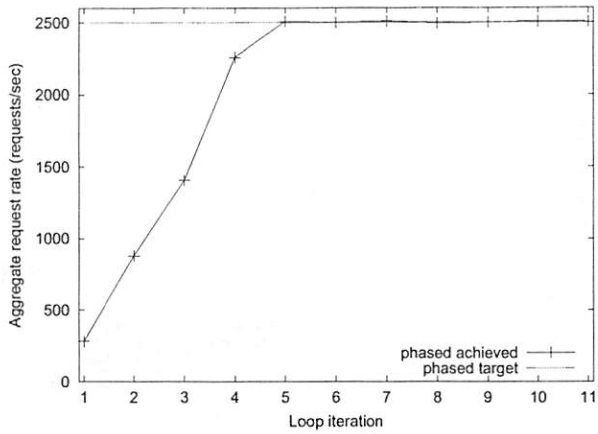


(b) Number of LUs

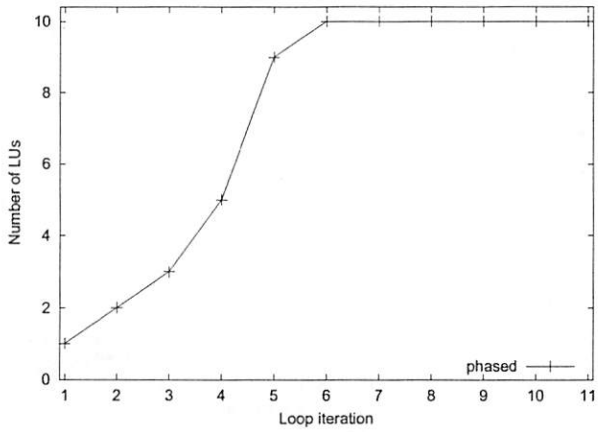
Figure 4: (a) Target and achieved average request rates at each iteration of the loop for the random-access workloads with a target aggregate request rate of 1250 req/sec. (b) Number of LUs used in each iteration.

4.3.2 Phased workloads

We also ran experiments where groups of streams had correlated ON/OFF periods. In these experiments, we used two stream groups, with all of the streams in the same group active simultaneously and only one group active at any time. Each group has an IOPS target of 2500 requests per second during its ON period, requiring all 10 LUs available on the disk array. Clearly, the storage system could not support the workload if both of the stream groups were active at the same time, but since the groups become active alternately, it is possible



(a) Average request rate



(b) Number of LUs

Figure 5: (a) Target and achieved average request rates at each iteration of the loop for the phased random-access workloads with two correlated stream groups with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used in each iteration.

for the storage system to support the workload. Figure 5 shows the average request rate achieved. We can see that the behavior of this workload is similar to the earlier always ON workload.

We now look at the distribution of the stores across the LUs. There are 100 stores in total; 50 in each group. What we expect is that each of the 10 LUs will end up containing 5 stores from group 1 and 5 stores from group 2. The imbalance of an LU is therefore the absolute value of the difference between the number of group 1 and group 2 stores on that LU. The *relative imbalance* over the entire storage system is then the sum of the imbalance of each LU divided by the number of LUs. In a balanced system, this metric should converge to zero. Figure 6 illustrates the relative imbalance for the phased workload. This figure shows that the solver correctly puts an equal number of stores from each group on each

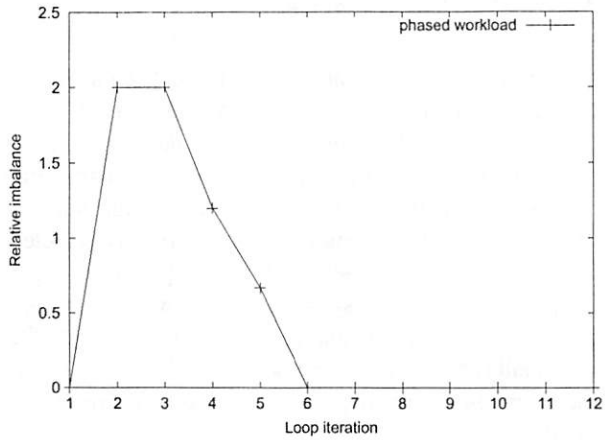


Figure 6: Relative imbalance of the two stream groups over the storage system for the phased workload.

LU for the phased workload; the imbalance goes to zero once the storage design has sufficient LUs.

4.4 PostMark

We ran the PostMark benchmark with a varying number of simultaneously active processes, which allows us to see the effect of different load levels on the behavior of the loop. Unlike the previous workloads, which issued requests at a fixed rate when correctly provisioned, PostMark is designed to issue requests at the peak rate the I/O configuration can sustain, given sufficient resources at the clients. By using multiple PostMark processes, we can effectively simulate greater client resources, and a higher load. In order to determine what the achievable performance was in practice, we first ran a set of experiments with the PostMark filesystem split over a varying number of LUs. Figure 7 shows how the PostMark transaction rates change as a function of the number of LUs and processes used. As can be seen, the system is limited primarily by the number of LUs. In all cases, the performance continues to increase as resources are added, although with diminishing returns. We presume that the performance will eventually level off, due to host software limitations, but we did not observe this for any except the one process case.

Ideally, Hippodrome would exhibit two properties with this workload. First, it should converge to a stable number of LUs, and not keep trying to indefinitely expand its resources. Second, the final system should be near the inflection point of the performance curve: i.e., increasing the number of LUs beyond this point would not result in significant performance increases. Table 3 shows that Hippodrome satisfies both of these properties, converging in all cases to a system that has performance close to the maximum achievable, using a reason-

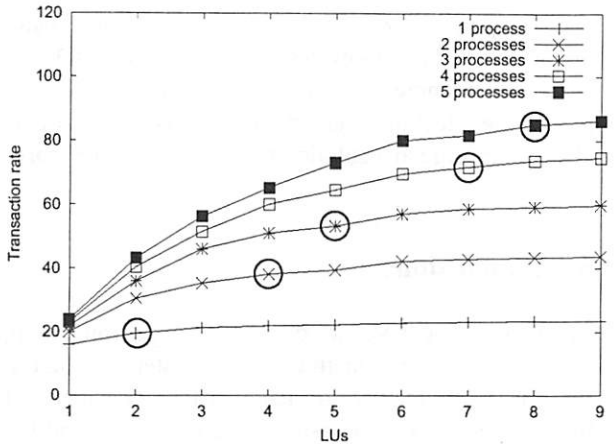


Figure 7: PostMark transaction rate as a function of number of LUs and processes used. The circles indicate the number of LUs/performance level to which Hippodrome converged.

number of processes	LUs	rate achieved	iterations	time
1	2	84%	2	1:13
2	4	87%	4	2:20
3	5	89%	5	2:53
4	7	96%	5	2:57
5	8	99%	5	2:59

Table 3: LUs, transaction rate achieved (as a percentage of the maximum observed for nine LUs), number of loop iterations and time to converge (hours and minutes) for the PostMark workload under varying numbers of processes.

able number of resources.

The wall clock time required for the loop to converge depends upon the number of iterations. Within each iteration, the majority of the time is spent copying the data from a master copy to the correct location in the new design, and takes between 20 and 40 minutes for the 50GB dataset, depending upon the number of LUs involved. The remaining time in each iteration is spent running the application, which takes roughly five to ten minutes. The analysis and the solver take a few minutes.

4.5 Summary

The experiments show that, for all workloads explored, Hippodrome satisfies the experimental goals. First, the system converges to the correct number of LUs in only a small number of loop iterations: at most four or five, and sometimes only one or two. Second, the designs that the system converges on are correctly provisioned; that is, the storage system contains the minimum number of LUs capable of supporting the offered workload. Finally, Hippodrome can leverage initial performance estimates (even inaccurate ones) to find the correct storage configuration more quickly.

These properties mean that Hippodrome can be used to perform storage system configuration automatically. The system administrators need only provide capacity information on the workload, and can then let Hippodrome handle the details of configuring the rest of the system resources, with the expectation that this configuration will happen in an efficient manner. In particular, administrators do not have to invest time and effort in the difficult task of deciding how to lay out the storage design; nor do they have to worry about whether the system will be able to support the application workload.

In the future, we would like to experiment with using more complex enterprise-scale workloads, such as a large database system. For such workloads, it is more difficult to tell if the loop did the "right thing", as we cannot easily *a priori* tell how good the design is, unlike the random-access and Postmark workloads. The applicability of Hippodrome to such systems is an ongoing research effort.

5 Related work

The EMC Symmetrix [14] and HP SureStore E XP512 Disk Arrays [19] support configuration adaptation to handle over-utilized LUs. They monitor LU utilization and use thresholds, set by the administrator, to trigger load-balancing via data migration within the array. The drawback is that they are unable to predict whether the move will be an improvement. Hippodrome's use of performance models allows it to evaluate whether a proposed migration would conflict with an existing workload.

HP's AutoRAID disk array [33] supports moving data between RAID5 and RAID1. AutoRAID keeps current data in RAID1 (since it has better performance), and uses an LRU policy based on write rate and capacity to migrate infrequently accessed data to RAID5, which has higher capacity. Hippodrome correctly places data based on the usage patterns, and expands the storage system if necessary to support increases in the workload.

Teradata [9] is a commercial parallel shared-nothing database that uses a hash on the primary index of a database table to statically partition the table across cluster nodes. This data placement allows data parallelism and improves the load balance. In contrast, Hippodrome dynamically reassigns stores based on observed device utilizations.

River [8] is a cluster-based I/O architecture that uses credit-based back pressure and graduated declustering

(GD) to distribute work in a manner proportional to the speed of the recipient nodes. However, River requires modifying the application, and it makes short-term load-balancing decisions, and does not handle long-term changes in the workload. Conversely, Hippodrome makes long-term decisions and does not require application modification.

A few other, automated tools exist that are useful to administrators of enterprise class systems. The AutoAdmin index selection tool [12] can automatically "design" a suitable set of indexes, given an input workload of SQL queries. It has a component that intelligently searches the space of possible indexes, similar to Hippodrome's design component, and an evaluation component (model, in Hippodrome terms) to determine the effectiveness of a particular selection based on the estimates from the query optimizer.

LEO, IBM DB2's "learning optimizer" [29], uses a feedback loop to enhance query optimization performance estimates based on observed past performance. It monitors previously executed queries and compares the optimizer's cost estimates with the actual performance at each step in the query execution plan, and then adjusts the cost estimates and statistics that may be used in future query optimizations. Although it does not currently do so, Hippodrome could use such feedback from observed system performance to improve the quality of its storage device performance models.

Océano [7] focuses on managing an e-business computing utility without human intervention, automatically allocating and configuring servers and network interconnections in a data center. It uses simple metrics for performance such as number of active connections and overall response time; it is similar in nature to the automatic loop in Section 2 in its management of compute and network resources.

Muse [11] controls server allocation and energy-conscious, adaptive resource provisioning tool for Internet hosting centers. It is also based on an iterative loop, like Hippodrome, but it focuses on allocating computational resources. Its resource allocation framework is based on an economic model that factors in the trade-offs between the service quality and the cost.

Existing solutions to the file assignment problem [13, 34] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The file allocation schemes described in [16, 28] will automatically determine an optimal stripe width for files, and stripe those files over a set of homogeneous disks. They then balance the load on those files based on a form of "hotspot" analysis, and swapping file blocks between "hot" and "cold" disks. Hippodrome can expand or contract the set of devices used, supports RAID systems, uses far more sophisticated performance models to predict the effect of system modifications, and will iteratively converge to a solution which supports the workload.

6 Conclusions

In this paper we have introduced the Hippodrome loop, our approach to automating storage system configuration. Hippodrome uses an iterative loop consisting of three stages: *analyze workload*, *design system*, and *implement design*. The components that implement these stages handle the problem of summarizing a workload, choosing which devices to use and how their parameters should be set, assigning the workload to the devices, and implementing the design by setting the device parameters and migrating the existing system to the new design.

We have shown that for the problem of storage system configuration, the Hippodrome loop satisfies two important properties:

- **Rapid convergence**: The loop converges in a small number of iterations to the final system design.
- **Correct resource allocation**: The loop allocates close to the minimal amount of resources necessary to support the workload.

We have demonstrated these properties using fixed-size, random-access workloads as well as the PostMark filesystem benchmark.

7 Ongoing and future work

We are currently extending Hippodrome to automatically manage the ongoing evolution of a storage system. Production systems evolve in time to handle device failures, changes in the workload, devices becoming obsolete, or new devices and workloads being added. Hippodrome should be able to detect and respond to these changes to keep the system appropriately provisioned and configured at all times. Preliminary results, using workloads similar to those described here, are promising. Using Hippodrome for on-line storage management also opens interesting research questions in controlling and/or maintaining quality of service, during both normal operation and while migration is taking place.

In the future, we plan to extend this work in several ways. First, we will investigate the sensitivity of the Hippodrome loop to the quality of its components. For in-

stance, what information must be captured in the analysis stage to sufficiently specify the performance requirements of the workload [24]? What is the sensitivity of the loop to the quality of the model component's predictions or to the quality of the solutions generated by the design component? In addition, we will continue to investigate how to build better loop components, for example, higher accuracy models, and a solver that minimizes store motion across iterations.

Second, we plan to experiment with complex enterprise-scale applications, highly variable workloads with load spikes, and workloads with natural load cycles (e.g. daily backups or monthly reports). How can we tell how well Hippodrome does for more complex workloads, given the difficulty of provisioning such large systems? How should the analysis infrastructure differentiate transient load spikes from workload growth trends? Furthermore, how should Hippodrome incorporate information about the cyclic nature of the workload to support all operational modes, while not grossly over-provisioning the system?

Third, we plan to extend Hippodrome to manage very large scale, heterogeneous and widely distributed storage systems. The experiments in this paper explored workloads that could fit onto a single disk array. How well does Hippodrome perform for workloads that require multiple disk arrays? How well does Hippodrome handle multiple types of disk arrays?

Additional research questions include the following:

- How well does Hippodrome interact with optimizations at the application level, which may result in changes to the I/O workload? For instance, how would the automated storage loop interact with database systems that automatically create indices as needed [12] or tune query plans based on observed performance [29]?

- How does Hippodrome fit into an overall end-to-end optimization scheme? For instance, how should Hippodrome cooperate with other solutions for storage area network design [32] or quality of service-preserving online migration [26]?

- What are the right evaluation metrics for the effectiveness of the automated loop? In addition to convergence time, potential metrics might be the number of errors, the overall system cost, the system performance, and the cost/performance tradeoff. An additional meta-level question is how to define "goodness" for each of these metrics.

8 Acknowledgements

We are grateful to our shepherd, Jeff Chase, for his invaluable comments, which improved our paper tremendously. We would like to thank John Wilkes for his comments on this paper, and Aaron Brown, David Oppenheimer, Dave Patterson, Arif Merchant and Erik Riedel for their comments on a previous version of this paper.

References

[1] N. Allen. Don't waste your storage dollars: what you need to know. Research note, Gartner Group, March 2001.

[2] G. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, Nov. 2001.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL–SSP–2001–4, HP Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[4] E. Anderson, J. Hall, J. Hartline, M.Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. In *5th Workshop on Algorithm Engineering (WAE2001)*, University of Aarhus, Denmark, Aug. 2001.

[5] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: an approach to solving the workload and device configuration problem. Technical Report HPL–SSP–2001–5, HP Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[6] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. Selecting RAID levels for disk arrays. In *Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Océano – SLA based management of a computing utility. In *Integrated Network Management VII*, May 2001.

[8] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *6th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'99)*, pages 10–22, Atlanta, GA, May 1999.

[9] C. Ballinger. Teradata database design 101: a primer on Teradata physical database design and its advantages. Technical note, NCR/Teradata, May 1998.

[10] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *1st Workshop on Software and Performance (WOSP'98)*, pages 199–207, Santa Fe, NM, Oct 1998.

[11] J. Chase, D. Anderson, P. Thakar, and A. Vahdat. Managing energy and server resources in hosting centers. In *18th ACM Symposium on Operating System Principles (SOSP'01)*, pages 103–116, Chateau Lake Louise, Banff, Canada, October 2001.

[12] S. Chaudhuri and V. Narasayya. AutoAdmin "What-if" index analysis utility. In *SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.

[13] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

[14] EMC Corporation. *EMC ControlCenter Product Description Guide*, 2000. Pub. No. 01748–9103.

[15] W. Fernandez and G. Lueker. Bin packing can be solved within 1+ε in linear time. *Combinatorica*, 1(4):349–55, 1981.

[16] P. Z. G. Weikum and P. Scheuermann. Dynamic file allocation in disk arrays. In *SIGMOD Conference*, pages 406–415, 1991.

[17] M. Garey and D. Johnson. *Computing and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[18] H. Group. Trends in e-business outsourcing and the rise of the managed hosting model. White paper, www.exodus.com, January 2001.

[19] Hewlett-Packard Company. *HP SureStore E Auto LUN XP User's guide*, August 2000. Pub. No. B9340–90900.

[20] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 - Advanced User's Guide*, December 2000.

[21] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, Dec. 1974.

[22] J. Katcher. Postmark: a new file system benchmark. Technical report TR-3022, Network Appliances, Oct 1997.

[23] C. Kenyon. Best-fit bin-packing with random order. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[24] Z. Kurmas, K. Keeton, and R. Becker-Szendy. I/O Workload Characterization. In *Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Jan 2001.

[25] E. Lamb. Hardware spending spatters. *Red Herring*, pages 32–33, June 2001.

[26] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST)*, Jan 2002.

[27] W. C. Preston. Using gigabit ethernet to backup six terabytes. In *12th Systems Administration Conference (LISA '98)*, pages 87–95, Boston, Massachusetts, 1998.

[28] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB Journal: Very Large Data Bases*, 7(1):48–66, 1998.

[29] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *27th Intl. Conference on Very Large Data Bases (VLDB)*, 2001.

[30] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS 2001)*, Cincinnati, OH, Aug. 2001.

[31] A. Veitch, K. Keeton, M. Spasojevic, and J. Wilkes. The Rubicon workload characterization tool. Technical report, HP Laboratories, April 2001. http://www.hpl.hp.com/SSP/.

[32] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. In *Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[33] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, Feb. 1996.

[34] J. Wolf. The placement optimization program: a practical solution to the disk file assignment problem. In *ACM SIGMETRICS Conference*, pages 1–10, Berkeley, CA, May 1989.

Selecting RAID levels for disk arrays

Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez and John Wilkes
Storage and Content Distribution Department,
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 95014
{anderse,swaram,aveitch,galvarez,wilkes}@hpl.hp.com

Abstract

Disk arrays have a myriad of configuration parameters that interact in counter-intuitive ways, and those interactions can have significant impacts on cost, performance, and reliability. Even after values for these parameters have been chosen, there are exponentially-many ways to map data onto the disk arrays' logical units. Meanwhile, the importance of correct choices is increasing: storage systems represent an growing fraction of total system cost, they need to respond more rapidly to changing needs, and there is less and less tolerance for mistakes. We believe that automatic design and configuration of storage systems is the only viable solution to these issues. To that end, we present a comparative study of a range of techniques for programmatically choosing the RAID levels to use in a disk array.

Our simplest approaches are modeled on existing, manual rules of thumb: they "tag" data with a RAID level before determining the configuration of the array to which it is assigned. Our best approach simultaneously determines the RAID levels for the data, the array configuration, and the layout of data on that array. It operates as an optimization process with the twin goals of minimizing array cost while ensuring that storage workload performance requirements will be met. This approach produces robust solutions with an average cost/performance 14–17% better than the best results for the tagging schemes, and up to 150–200% better than their worst solutions.

We believe that this is the first presentation and systematic analysis of a variety of novel, fully-automatic RAID-level selection techniques.

1 Introduction

Disk arrays are an integral part of high-performance storage systems, and their importance and scale are growing as continuous access to information becomes critical to the day-to-day operation of modern business.

Before a disk array can be used to store data, values for many configuration parameters must be specified: achieving the right balance between cost, availability, and application performance needs depends on many correct decisions. Unfortunately, the tradeoffs between the choices are surprisingly complicated. We focus here on just one of these choices: which RAID level, or data-redundancy scheme, to use.

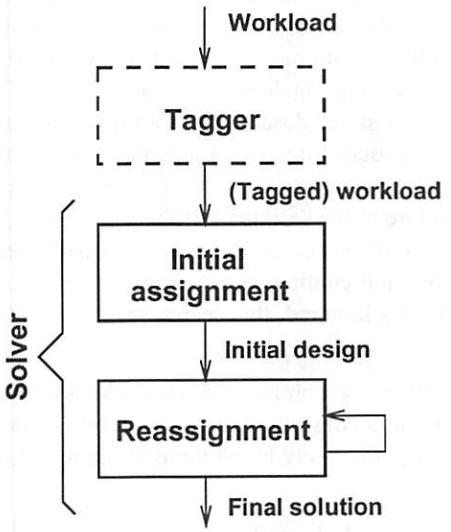


Figure 1: The decision flow in making RAID level selections, and mapping stores to devices. If a tagger is present, it irrevocably assigns a RAID level to each store before the solver is run; otherwise, the solver assigns RAID levels as it makes data layout decisions. Some variants of the solver allow revisiting this decision in a final reassignment pass; others do not.

The two most common redundancy schemes are RAID 1/0 (striped mirroring), where every byte of data is kept on two separate disk drives, and striped for greater I/O parallelism, and RAID 5 [20], where a single parity block protects the data in a stripe from disk drive failures. RAID 1/0 provides greater read performance and failure tolerance—but requires almost twice as many disk drives to do so. Much prior work has studied the properties of different RAID levels (e.g., [2, 6, 20, 11]).

Disk arrays organize their data storage into *Logical Units*, or LUs, which appear as linear block spaces to their clients. A small disk array, with a few disks, might support up to 8 LUs; a large one, with hundreds of disk drives, can support thousands. Each LU typically has a given RAID level—a redundancy mapping onto one or more underlying physical disk drives. This decision is made at LU-creation time, and is typically irrevocable: once the LUhas been formatted, changing its RAID level requires copying all the data onto a new LU.

Following previous work [7, 25], we describe the workloads to be run on a storage system as sets of *stores* and *streams*. A store is a logically contiguous array of bytes, such as a file system or a database table, with a size typically measured in gigabytes; a stream is a set of access patterns on a store, described by attributes such as request rate, request size, inter-stream phasing information, and sequentiality. A RAID level must be decided for each store in the workload; if there are k RAID levels to choose from and m stores in the workload, then there are k^m feasible configurations. Since $k \geq 2$ and m is usually over a hundred, this search space is too large to explore exhaustively by hand.

Host-based logical volume managers (LVMs) complicate matters by allowing multiple stores to be mapped onto a single LU, effectively blending multiple workloads together.

There is no single best choice of RAID level: the right choice for a given store is a function of the access patterns on the store (e.g., reads versus writes; small versus large; sequential versus random), the disk array's characteristics (including optimizations such as write buffer merging [23], segmented caching [26], and parity logging [21]), and the effects of other workloads and stores assigned to the same array [18, 27].

In the presence of these complexities, system administrators are faced with the tasks of (1) selecting the type and number of arrays; (2) selecting the size and RAID level for each LU in each disk array; and (3) placing stores on the resulting LUs. The administrators' goals are operational in nature, such as minimum cost, or maximum reliability for a given cost—while satisfying the performance requirements of client applications. This is clearly a very difficult task, so manual approaches apply rules of thumb and gross over-provisioning to simplify the problem (e.g., "stripe each database table over as many RAID 1/0 LUs as you can"). Unfortunately, this paper shows that the resulting configurations can cost as much as a factor of two to three more than necessary. This matters when the cost of a large storage system can easily be measured in millions of dollars and represents more than half the total system hardware cost. Perhaps even more important is the uncertainty that surrounds a manually-designed system: (how well) will it meet its performance and availability goals?

We believe that automatic methods for storage system design [1, 5, 7, 4] can overcome these limitations, because they can consider a wider range of workload interactions, and explore a great deal more of the search space than any manual method. To do so, these automatic methods need to be able to make RAID-level selection decisions, so the question arises: what is the best way to do this selection? This paper introduces a variety of approaches for answering this question.

The rest of the paper is organized as follows. In Section 2 we describe the architecture of our RAID level selection infrastructure. We introduce the schemes that operate on a per-store basis in Section 3, and in Section 4 we present a family of methods that simultaneously account for prior data placement and RAID level selection decisions. In Section 5, we compare all the schemes by doing experiments with synthetic and realistic workloads. We conclude in Sections 6 and 7 with a review of related work, and a summary of our results and possible further research.

2 Automatic selection of RAID levels

Our approach to automating storage system design relies on a *solver*: a tool that takes as input (1) a workload description and (2) information about the target disk array types and their configuration choices. The solver's output is a design for a storage system capable of supporting that workload.

In the results reported in this paper, we use our third-generation solver, *Ergastulum* [5] (prior solver generations were called Forum [7] and Minerva [1]). Our solvers are constraint-based optimization systems that use analytical and interpolation-based *performance models* [3, 7, 18, 23] to determine whether performance constraints are being met by a tentative design. Although such models are less accurate than trace-driven simulations, they are much faster, so the solver can rapidly evaluate many potential configurations.

As illustrated in Figure 1, the solver designs configurations for one or more disk arrays that will satisfy a given workload. This includes determining the array type and count, selecting the configuration for each LU in each array, and assigning the stores onto the LUs. In general, solvers rely on heuristics to search for the solution that minimizes some user-specified goal or objective. All experiments in this paper have been run with the objective

of minimizing the hardware cost of the system being designed, while satisfying the workload's performance requirements.

During the optional *tagging* phase, the solver examines each store, and tags it with a RAID level based on the attributes of the store and associated streams.

During the *initial assignment* phase, Ergastulum explores the array design search space by first randomizing the order of the stores, and then running a best-fit search algorithm [10, 15, 16] that assigns one store at a time into a tentative array design. Given two possible assignments of a store onto different LUs, the solver uses an externally-selected *goal function* to choose the "best" assignment. While searching for the best placement of a store, the solver will try to assign it onto the existing LUs, to purchase additional LUs on existing arrays, and to purchase additional arrays. A goal function that favors lower-cost solutions will bias the solver towards using existing LUs where it can.

At each assignment, the solver uses its performance models to perform *constraint checks*. These checks ensure that the result is a feasible, valid solution that can accommodate the capacity and performance requirements of the workload.

The *reassignment* phase of the solver algorithm attempts to improve on the solution found in the initial phase. The solver randomly selects a complete LU from the existing set, removes all the stores from it, and *reassigns* them, just as in the first phase. It repeats this process until every single LUs has been reassigned a few times (a configurable parameter that we set to 3). The reassignment phase is designed to help the solver avoid local minima in the optimization search space. This phase produces a near-optimal assignment of stores to LUs. For more details on the optimality of the assignments and on the operation of the solver, we refer the interested reader to [5].

2.1 Approaches to RAID level selection

We explore two main approaches to selecting a RAID level:

1. *Tagging* approaches: These approaches perform a pre-processing step to tag stores with RAID levels before the solver is invoked. Once tagged with a RAID level, a store cannot change its tag, and it must be assigned to an LU of that type. Tagging decisions consider each store and its streams in isolation. We consider two types of taggers: *rule-based*, which examine the size and type of I/Os; and *model-based*, which use performance models to make their decisions. The former tend to have many *ad hoc* parameter settings; the latter have fewer, but also need performance-related data for a particular disk array type. In some cases we use the same performance models as we later apply in the solver.

2. *Solver-based*, or *integrated*, approaches: These omit the tagging step, and defer the choice of RAID level until data-placement decisions are made by the solver. This allows the RAID level decision to take into account interactions with the other stores and streams that have already been assigned.

   We explored two variants of this approach: a *partially adaptive* one, in which the RAID level of an LU is chosen when the first store is assigned to it, and cannot subsequently be changed; and a *fully adaptive* variant, in which any assignment pass can revisit the RAID level decision for an LU at any time during its best-fit search. In both cases, the reassignment pass can still change the bindings of stores to LUs, and even move a store to an LU of a different RAID level.

   Neither variant requires any *ad hoc* constants, and both can dynamically select RAID levels. The fully adaptive approach has greater solver complexity and longer running times, but results in an exploration of a larger fraction of the array design search space.

Table 1 contrasts the four families of RAID level selection methods we studied.

We now turn to a detailed description of these approaches.

3 Tagging schemes

Tagging is the process of determining, for each store in isolation, the appropriate RAID level for it. The solver must later assign that store to an LU with the required RAID level. The tagger operates exactly once on each store in the input workload description, and its decisions are final. We followed this approach in previous work [1] because the decomposition into two separate stages is natural, is easy to understand, and limits the search space that must be explored when designing the rest of the storage system.

We explore two types of taggers: one type based on rules of thumb and the other based on performance models.

3.1 Rule-based taggers

These taggers make their decisions using rules based on the size and type of I/Os performed by the streams. This

Approach	*Goal functions*	*Solver*	*Summary*
Rule-based tagging	no change	simple	many constants, variable results
Model-based tagging	no change	simple	fewer constants, variable results
Partially-adaptive solver	special for initial assignment	simple	good results, limited flexibility
Fully-adaptive solver	no change	complex	good results, flexible but slower

Table 1: The four families of RAID-level selection methods studied in this paper. The two tagging families use either rule-based or model-based taggers. The model-based taggers use parameters appropriate for the array being configured. The fully adaptive family uses a substantially more complex solver than the other families. The *Goal functions* column indicates whether the same goal functions are used in both solver phases: initial assignment and reassignment. The *Summary* column provides an evaluation of their relative strengths and weaknesses.

is the approach implied by the original RAID paper [20], which stated, for example, that RAID 5 is bad for "small" writes, but good for "big" sequential writes. This approach leads to a large collection of device-specific constants, such as the number of seeks per second a device can perform, and device-specific thresholds, such as where exactly to draw the line between a "mostly-read" and a "mostly-write" workload. These thresholds could, in principle, be workload-independent, but in practice, we found it necessary to tune them experimentally to our test workloads and arrays, which means that there is no guarantee they will work as well on any other problem.

The rules we explored were the following. The first three taggers help provide a measure of the cost of the *laissez-faire* approaches. The remaining ones attempt to specify concrete values for the rules of thumb proposed in [20].

1. *random:* pick a RAID level at random.
2. *allR10:* tag all stores RAID 1/0.
3. *allR5:* tag all stores RAID 5.
4. *R5BigWrite:* tag a store RAID 1/0 unless it has "mostly" writes (the threshold we used was at least 2/3 of the I/Os), and the writes are also "big" (greater than 200 KB, after merging sequential I/O requests together).
5. *R5BigWriteOnly:* tag a store RAID 1/0 unless it has "big" writes, as defined above.
6. *R10SmallWrite:* tag a store RAID 5 unless it has "mostly" writes and the writes are "small" (i.e., not "big").
7. *R10SmallWriteAggressive:* as R10SmallWrite, but with the threshold for number of writes set to 1/10 of the I/Os rather than 2/3.

In practice, we found these rules needed to be augmented with an additional rule to determine if a store was *capacity-bound* (i.e., if space, rather than performance, was likely to be the bottleneck resource). A capacity-bound store was always tagged as RAID 5. This rule required additional constants, with units of bytes-per-second/GB and seeks-per-second/GB; these values had to be computed independently for each array. (Also, it is unclear what to do if an array can support different disk types with different capacity/performance ratios.)

We also evaluated each of these taggers without the capacity-bound rule. These variations are shown in the graphs in Section 5 by appending *Simple* to each of the tagger names.

3.2 Model-based taggers

The second type of tagging methods we studied used array-type-specific performance models to estimate the effect of assigning a store to an LU, and made a selection based on that result.

The first set of this type use simple performance models that predict the number of back-end I/Os per second (*IOPS*) that will result from the store being tagged at each available RAID level, and then pick the RAID level that minimizes that number. This removes some *ad hoc* thresholds such as the size of a "big" write, but still requires array-specific constants to compute the IOPS estimates. These taggers still need the addition of the capacity-bound rule to get decent results. The IOPS-based taggers we study are:

8. *IOPS:* tag a store RAID 1/0 if the estimated IOPS would be smaller on a RAID 1/0 than on a RAID 5 LU. Otherwise tag it as RAID 5.
9. *IOPS-disk:* as *IOPS* except the IOPS estimates are divided by the number of disks in the LU, resulting in a per-disk IOPS measure, rather than a per-LU measure. The intent is to reflect the potentially different number of disks in RAID 1/0 and RAID 5 LUs.
10. *IOPS-capacity:* as *IOPS* except the IOPS estimates are multiplied by the ratio of raw (unprotected) ca-

pacity divided by effective capacity. This measure factors in the extra capacity cost associated with RAID 1/0.

The second set of model-based taggers use the same performance models that needed to be constructed and calibrated for the solver anyway, and does not depend on any *ad hoc* constants. These taggers use the models to compute, for each available RAID level, the percentage changes in the LU's utilization and capacity that will result from choosing that level, under the simplifying assumption that the LU is dedicated solely to the store being tagged. We then form a 2-dimensional vector from these two results, and then pick the RAID level that minimizes:

11. *PerfVectLength:* the length (L_2 norm) of the vector;
12. *PerfVectAvg:* the average magnitude (L_1 norm) of the components;
13. *PerfVectMax:* the maximum component (L_∞ norm);
14. *UtilizationOnly:* just the utilization component, ignoring capacity.

4 Solver-based schemes

When we first tried using the solver to make all RAID-level decisions, we discovered it worked poorly for two related reasons:

1. The solver's goal functions were cost-based, and using an existing LU is always cheaper than allocating a new one.
2. The solver chooses a RAID level for a new LU when it places the first store onto it – and a 2-disk RAID 1/0 LU is always cheaper than a 3- or more-disk RAID 5 LU. As a result, the solver would choose a RAID 1/0 LU, fill it up, and then repeat this process, even though the resulting system would cost more because of the additional disk space needed for redundancy in RAID 1/0. (Our tests on the FC-60 array (described in Section 5.2) did not have this discrepancy because we arranged for the RAID 1/0 and RAID 5 LUs to contain six disks each, to take best advantage of the array's internal bus structure.)

We explored two options for addressing these difficulties. First, we used a number of different initial goal functions that ignored cost, in the hope that this would give the reassignment phase a better starting point. Second, we extended the solver to allow it to change the RAID level of an LU even after stores had been assigned to it.

We refer to the first option as *partially-adaptive*, because it can change the RAID level associated with an individual store—but it still fixes an LU's RAID level when the first store is assigned to it. Adding another goal function to the solver proved easy, so we tried several in a search for one that worked well. We refer to the second option as *fully-adaptive* because the RAID level of the store and the LUs can be changed at almost any time. It is more flexible than the partially-adaptive one, but required more extensive modifications to the solver's search algorithm.

4.1 Partially-adaptive schemes

The partially-adaptive approach works around the problem of the solver always choosing the cheaper, RAID 1/0 LUs, by ignoring cost considerations in the initial selection – thereby avoiding local cost-derived minima – and reintroducing cost in the reassignment stage. By allowing more LUs with more-costly RAID levels, the reassignment phase would have a larger search space to work within, thereby producing a better overall result.

Even in this scheme, the solver still needs to decide whether a newly-created LU should be labeled as RAID 5 or RAID 1/0 during the initial assignment pass. It does this by means of a *goal function*. The goal function can take as input the performance, capacity, and utilization metrics for all the array components that would be involved in processing accesses to the store being placed into the new LU. We devised a large number of possible initial goal functions, based on the combinations of these metrics that seemed reasonable. While it is possible that there are other, better initial goal functions, we believe we have good coverage of the possibilities. Here is the set we explored:

1. *allR10:* always use RAID 1/0.
2. *allR5:* always use RAID 5.
3. *AvgOfCapUtil:* minimize the average of capacities and utilizations of all the disks (the L_1 norm).
4. *LengthOfCapUtil:* minimize the sum of the squares of capacities and utilizations (the L_2 norm) of all the disks.
5. *MaxOfCapUtil:* minimize the maximum of capacities and utilizations of all the disks (the L_∞ norm).
6. *MinAvgUtil:* minimize the average utilizations of all the array components (disks, controllers and internal buses).

7. *MaxAvgUtil:* maximize the average utilizations of all the array components (disks, controllers and internal buses).
8. *MinAvgΔUtil:* minimize the arithmetic mean of the change in utilizations of all the array components (disks, controllers and internal buses).
9. *MinAvgΔUtilPerRAIDdisk:* as with scheme (8), but first divide the result by the number of *physical* disks used in the LU.
10. *MinAvgΔUtilPerDATAdisk:* as with scheme (8), but first divide the result by the number of *data* disks used in the LU.
11. *MinAvgΔUtilTimesRAIDdisks:* as with scheme (8), but first multiply the result by the number of *physical* disks used in the LU.
12. *MinAvgΔUtilTimesDATAdisks:* as with scheme (8), but first multiply the result by the number of *data* disks used in the LU.

The intent of the various disk-scaling schemes (9–12) was to explore ways of incorporating the size of an LU into the goal function.

Goal functions for the reassignment phase make minimal system cost the primary decision metric, while selecting the right kind of RAID level is used as a tie-breaker. As a result, there are fewer interesting choices of goal function during this phase, and we used just two:

1. *PriceThenMinAvgUtil:* lowest cost, ties resolved using scheme (6).
2. *PriceThenMaxAvgUtil:* lowest cost, ties resolved using scheme (7).

During our evaluation, we tested each of the reassignment goal functions in combination with all the initial-assignment goal functions listed above.

4.2 Fully-adaptive approach

As we evaluated the partly-adaptive approach, we found several drawbacks that led us to try the more flexible, fully-adaptive approach:

- After the goal functions had become cost-sensitive in the reassignment phase, new RAID 5 LUs would not be created. Solutions would suffer if there were too few RAID 5 LUs after initial assignment.
- It was not clear how well the approach would extend to more than two RAID levels.
- Although we were able to achieve good results with the partially-adaptive approach, the reasons for the results were not always obvious, hinting at a possible lack of robustness.

To address these concerns, we extended the search algorithm to let it dynamically switch the RAID level of a given LU. Every time the solver considers assigning a store to an LU (that may already have stores assigned to it), it evaluates whether the resulting LU would be better off with a RAID 1/0 or RAID 5 layout.

The primary cost of the fully-adaptive approach is that it requires more CPU time than the partially-adaptive approach, which did not revisit RAID-level selection decisions. In particular, the fully-adaptive approach roughly doubles the number of performance-model evaluations, which are relatively expensive operations. But fully-adaptive approach has several advantages: the solver is no longer biased towards a given RAID level, because it can identify the best choice at all stages of the assignment process. Adding more RAID levels to choose from is also possible, although the total computation time grows roughly linearly with the number of RAID levels. And there no longer is a need for a special goal function during the initial assignment phase.

Our experiments showed that, with two exceptions, the *PriceThenMinAvgUtil* and *PriceThenMaxAvgUtil* goal functions produced identical results for all the fully-adaptive schemes. Each was better for one particular workload; we selected *PriceThenMaxAvgUtil* for our experiments, as it resulted in the lowest average cost. We found that it was possible to improve the fully-adaptive results slightly (so that they always produced the lowest cost) by increasing the number of reassignment passes to 5, but we did not do so to keep the comparison with the partially-adaptive solver as fair as possible.

5 Evaluation

In this section, we present an experimental evaluation of the effectiveness of the RAID level selection schemes discussed above.

We took workload specifications from [1] and from traces of a validated TPC-D configuration. We used the Ergastulum solver to design storage systems to support these workloads, ensuring for each design that the performance and capacity needs of the workload would be met. To see if the results were array-specific, we constructed designs for two different disk array types.

The primary evaluation criterion for the RAID-level selection schemes was the cost of the generated configurations, because our performance models [3, 18] predicted

Workload	*Capacity*	*#stores*	*#streams*	*Access size*		*Run count*		*%reads*
filesystem	0.09 TB	140	140	20.0 KB	(±13.8)	2.6	(±1.3)	64.2%
scientific	0.19 TB	100	200	640.0 KB	(±385.0)	93.5	(±56.6)	20.0%
oltp	0.19 TB	194	182	2.0 KB	(±0.0)	1.0	(±0.0)	66.0%
fs-light	0.16 TB	170	170	14.8 KB	(±7.3)	2.1	(±0.7)	64.1%
tpcd30	0.05 TB	316	224	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd30-2x	0.10 TB	632	448	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd30-4x	0.20 TB	1264	896	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd300-1	1.95 TB	911	144	53.5 KB	(±12.8)	1.13	(±0.1)	98.3%
tpcd300-5	1.95 TB	935	374	49.1 KB	(±10.6)	1.23	(±1.9)	92.7%
tpcd300-7	1.95 TB	941	304	51.1 KB	(±10.7)	1.12	(±0.1)	95.0%
tpcd300-9	1.95 TB	933	399	49.8 KB	(±10.6)	1.20	(±1.9)	85.6%
tpcd300-10	1.95 TB	910	321	45.3 KB	(±12.3)	1.28	(±2.2)	80.3%

Table 2: Characteristics of workloads used in experiments. "Run count" is the mean number of consecutive sequential accesses made by a stream. Thus workloads with low run counts (*filesystem*, *oltp*, *fs-light*) have essentially random accesses, while workloads with high run counts (*scientific*) have sequential accesses. *tpcd* has both streams with random and sequential accesses. The access size and run count columns list the mean and (standard deviation) for these values across all streams in the workload.

that all the generated solutions would support the workload performance requirements. The secondary criterion was the CPU time taken by each approach.

We chose not to run the workloads on the target physical arrays because it was not feasible. First, we did not have access to the applications used for some of the workloads—just traces of them running. Second, there were too many of them. We evaluated over a thousand configurations for the results presented; many of the workloads run for hours. Third, some of the resulting configurations were too large for us to construct. Fortunately, previous work [1] with the performance models we use indicated that their performance predictions are sufficiently accurate to allow us to feel confident that our comparisons were fair, and that the configurations designed would indeed support the workloads.

5.1 Workloads

To evaluate the RAID-level selection schemes, we used a number of different workloads that represented both traces of real systems and models of a diverse set of applications: an active file system (*filesystem*), a scientific application (*scientific*), an on-line transaction processing benchmark (*oltp*), a lightly-loaded filesystem (*fs-light*), a 30 GB TPC-D decision-support benchmark, running three queries in parallel until all of them complete (*tpcd30*), the *tpcd30* workload duplicated (as if they were independent, but simultaneous runs) 2 and 4 times (*tpcd30-2x* and *tpcd30-4x*), and the most I/O-intensive queries (i.e., 1, 5, 7, 9 and 10) of the 300 GB TPC-D benchmark run one at a time on a validated configuration (*tpcd300-query-N*).

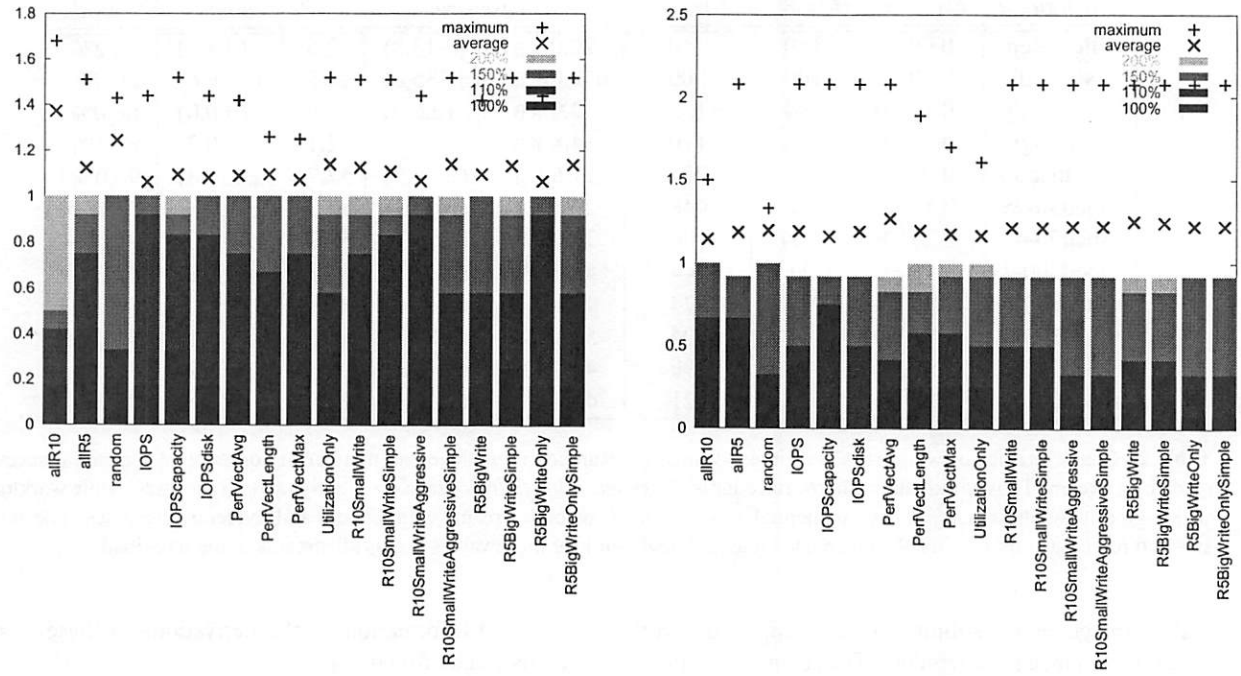
Table 2 summarizes their performance characteristics. Detailed information on the derivations of these workloads can be found in [1].

5.2 Disk arrays

We performed experiments using two of the arrays supported by our solver: the Hewlett-Packard SureStore Model 30/FC High Availability Disk Array (FC-30, [12]) and the Hewlett-Packard SureStore E Disk Array FC-60 (FC-60, [13]), as these are the ones for which we have calibrated models.

The FC-30 is characteristic of a low-end, stand-alone disk array of 3–4 years ago. An FC-30 has up to 30 disks of 4 GB each, two redundant controllers (to survive a controller failure) and 60 MB of battery-backed cache (NVRAM). Each of the two array controllers is connected to the client host(s) over a 1 Gb/s FibreChannel network. Our FC-30 performance models [18] have an average error of ±6% and a worst-case error of ±20% over a reasonable range of LU sizes.

The FC-60 is characteristic of modern mid-range arrays. An FC-60 array can have up to 60 disks, placed in up to six disk enclosures. Each of the two array controllers is connected to the client host(s) over a 1 Gb/s FibreChannel network. Each controller may have up to 512 MB of NVRAM. The controller enclosure contains a backplane bus that connects the controllers to the disk enclosures, via six 40 MB/s ultra-wide SCSI busses. Disks of up to 72 GB can be used, for a total unprotected capacity of 4.3 TB. Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. Our interpolation-based FC-60 performance models [3] have an average error of about 10% over a fairly wide range of configurations.



(a) **FC-30 array**

(b) **FC-60 array**

Figure 2: Tagger results for the FC-30 and FC-60 disk arrays. The results for each tagger are plotted within a single bar of the graph. Over all workloads, the bars show the proportion of time each tagger resulted in a final solution with the lowest cost (as measured over all varieties of RAID level selection), within 110% of the lowest cost, within 150% of the lowest cost and within 200% of the cost. The taller and darker the bar, the better the tagger. Above each bar, the points show the maximum (worst) and average results for the tagger, as a multiple of the best cost. The *allR10* and *allR5* taggers tag all stores as RAID 1/0 or RAID 5 respectively. The *random* tagger allocates stores randomly to either RAID level. The *IOPS* models are based on very simple array models. The *PerfVect...* and *UtilizationOnly* taggers are based on the complete analytical models as used by the solver. The remaining taggers are rule-based.

5.3 Comparisons

As described above, the primary criteria for comparison for all schemes is that of total system cost.

5.3.1 Tagger results

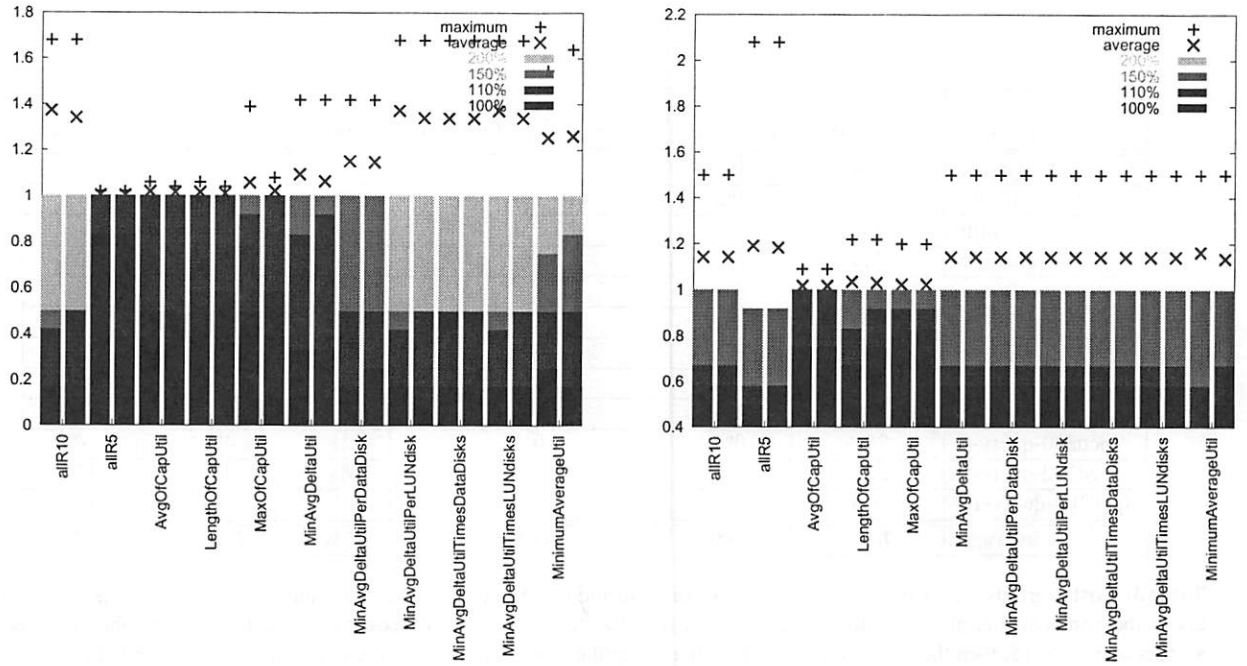
Figure 2 shows the results for each of the taggers for the FC-30 and FC-60 arrays. There are several observations and conclusions we can draw from these results.

First, there is no overall winner. Within each array type, it is difficult to determine what the optimal choice is. For instance, compare the *PerfVectMax* and *IOPS* taggers for the FC-30 array. *IOPS* has a better average result than *PerfVectMax*, but performs very badly on one workload (*filesystem*), whereas *PerfVectMax* is much better in the worst case. Depending on the user's expected range of workloads, either one may be the right choice.

When comparing results across array types, the situation is even less clear—the sets of best taggers for each array are completely disjoint. Hence, the optimal choice of RAID level varies widely from array to array, and no single set of rules seems to work well for all array types, even when a subset of all array-specific parameters (such as the test for capacity-boundedness) is used in addition.

Second, the results for the FC-60 are, in general, worse than for the FC-30. In large part, this is due to the relative size and costs of the arrays. Many of the workloads require a large number (more than 20) of the FC-30 arrays; less efficient solutions—even those that require a few more complete arrays—add only a small relative increment to the total price. Conversely, the same workloads for the FC-60 only require 2–3 arrays, and the relative cost impact of a solution requiring even a single extra array is considerable. Another reason for the increased FC-60 costs is that many of the taggers were hand-tuned for the FC-30 in an earlier series of experiments [1].

With a different array, which has very different perfor-



(a) **FC-30 array**

(b) **FC-60 array**

Figure 3: Partially-adaptive results for the FC-30 and FC-60 disk arrays. There are two bars for each initial assignment goal function: the one on the left uses the *PriceThenMinAvgUtil* reassignment goal function, the one on the right *PriceThenMaxAvgUtil*.

mance characteristics, the decisions as to what constitutes a "large write" become invalid. For example, consider the stripe size setting for each of the arrays we used. The FC-60 uses a default of 16 KB, whereas the FC-30 uses 64 KB, which results in different performance for I/O sizes between these values.

Third, even taggers based completely on the solver models perform no better, and sometimes worse, than taggers based only on simple rules. This indicates that tagging solutions are too simplistic; it is necessary to take into account the interactions between different streams and stores mapped to the same LU or array when selecting RAID levels. This can be done through the use of adaptive algorithms, as shown in the following sections.

5.3.2 Partially-adaptive results

Figure 3 shows results for each of the partially-adaptive rules for the FC-30 and FC-60 arrays. Our results show that the partly adaptive solver does much better than the tagging approaches. In particular, minimizing the average capacity and utilization works well for both arrays and all the workloads.

From the data, it is clear that *allR5* is the best partially-adaptive rule for the FC-30 but not for the FC-60. However the rules based on norms (*AvgOfCapUtil*, *MaxOfCapUtil* and *LengthOfCapUtil*) seem to perform fairly well for both arrays—an improvement over the tagging schemes. The family of partially-adaptive rules based on change in utilization seems to perform reasonably for the FC-30, but poorly for the FC-60—with one exception, *MinAvgΔUtilTimesDataDisks*, that performed as well as the norm-rules.

5.3.3 Fully-adaptive results

Tables 3 and 4 show, for each workload, the best results achieved for each family of RAID level selection methods. As can be seen, the fully-adaptive approach finds the best solution in all but one case, indicating that this technique better searches the solution space than the partly adaptive and tagging techniques. Although the fully-adaptive approach needs more modifications to the solver, a single goal function performs nearly perfectly on both arrays, and it is more flexible.

	Taggers			*Partly adaptive*		*Fully*
Workload	*PerfVectMax*	*IOPSdisk*	*R10SmallWriteAggressive*	*AllR5*	*AvgOfCapUtil*	*adaptive*
filesystem	1%	1%	1%	0%	0%	0%
filesystem-lite	1%	1%	3%	0%	0%	0%
oltp	1%	1%	1%	0%	0%	0%
scientific	2%	2%	2%	0%	2%	0%
tpcd30-1x	7%	8%	8%	0%	0%	0%
tpcd30-2x	4%	2%	2%	2%	2%	0%
tpcd30-4x	1%	44%	44%	2%	2%	2%
tpcd300-query-1	0%	0%	0%	0%	0%	0%
tpcd300-query-5	18%	4%	4%	0%	0%	0%
tpcd300-query-7	25%	0%	0%	0%	4%	0%
tpcd300-query-9	22%	4%	8%	0%	4%	0%
tpcd300-query-10	4%	8%	8%	0%	0%	0%
average	**7.2%**	**6.3%**	**6.8%**	**0.3%**	**1.2%**	**0.12%**

Table 3: Cost overruns for the best solution for each workload and RAID selection method for the FC-30 array. Values are in percent above the best cost over all results for that array—that is, if the best possible result cost $100, and the given method resulted in a system costing $115, then the cost overrun is 15%. Increasing the number of reassignment passes to 5 results in the fully-adaptive scheme being best in all cases; we do not report those numbers to present a fair comparison with the other schemes.

	Taggers			*Partly adaptive*		*Fully*
Workload	*FC60UtilizationOnly*	*IOPScapacity*	*allR10*	*AvgOfCapUtil*	*MaxOfCapUtil*	*adaptive*
filesystem	0%	44%	0%	0%	0%	0%
filesystem-lite	24%	0%	24%	0%	0%	0%
oltp	0%	2%	0%	0%	0%	0%
scientific	0%	108%	0%	0%	0%	0%
tpcd30-1x	12%	12%	0%	0%	0%	0%
tpcd30-2x	9%	9%	0%	0%	0%	0%
tpcd30-4x	7%	7%	0%	7%	0%	0%
tpcd300-query-1	7%	2%	50%	5%	0%	0%
tpcd300-query-5	61%	2%	37%	0%	9%	0%
tpcd300-query-7	32%	1%	0%	0%	20%	0%
tpcd300-query-9	36%	2%	10%	0%	0%	0%
tpcd300-query-10	12%	2%	50%	9%	0%	0%
average	**16.7%**	**15.9%**	**14.3%**	**1.75%**	**2.4%**	**0%**

Table 4: Cost overruns for the best solution for each workload and RAID selection method for the FC-60 array. All values are percentages above the best cost seen across all the methods.

5.3.4 CPU time comparison

The advantage of better solutions does not come without a cost: Table 5 shows that the CPU time to calculate a solution increases for the more complex algorithms, because they explore a larger portion of the search space. In particular, tagging eliminates the need to search any solution that uses an LU with a different tag, and makes selection of a new LU's type trivial when it is created, whereas both of the adaptive algorithms have to perform a model evaluation and a search over all of the LU types.

The fully-adaptive algorithm searches all the possibilities that the partially-adaptive algorithm does, and also looks at the potential benefit of switching the LU type on each assignment. It takes considerably longer to run. Even so, this factor is insignificant when put into context: our solver has completely designed enterprise storage systems containing $2–$5 million of storage equipment in under an hour of CPU time. We believe that the advantages of the fully-adaptive solution will outweigh its computation costs in almost all cases.

5.3.5 Implementation complexity

A final tradeoff that might be considered is the implementation complexity. The modifications to implement partially-adaptive schemes on the original solver took a few hours of work. The fully-adaptive approach took a few weeks of work. Both figures are for a person thoroughly familiar with the solver code. However, the fully-adaptive approach clearly gives the best results, and is independent of the devices and workloads being used; the development investment is likely to pay off very quickly in any production environment.

6 Related work

The published literature does not seem to report on systematic, implementable criteria for automatic RAID level selection. In their original paper [20], Patterson, Gibson and Katz mention some selection criteria for RAID 1 through RAID 5, based on the sizes of read and write accesses. Their criteria are high-level rules of thumb that apply to extreme cases, e.g., "if a workload contains mostly small writes, use RAID 1/0 instead of RAID 5". No attempt is made to resolve contradictory recommendations from different rules, or to determine threshold values for essential definitions like "small write" or "write-mostly". Simulation-based studies [2, 14, 17] quantify the relative strengths of different RAID levels (including some not mentioned in this paper), but do not derive general guidelines for choosing a RAID level for given access patterns.

The HP AutoRAID disk array [24] side-steps the issue by dynamically, and transparently, migrating data blocks between RAID 1/0 and RAID 5 storage as a result of data access patterns. However, the AutoRAID technology is not yet widespread, and even its remapping algorithms are themselves based on simple rules of thumb that could perhaps be improved (e.g., "put as much recently written data in RAID 1/0 as possible").

In addition to RAID levels, storage systems have multiple other parameters that system administrators are expected to set. Prior studies examined how to choose the number of disks per LU [22], and the optimal stripe unit size for RAID 0 [9], RAID 5 [8], and other layouts [19]. The RAID Configuration Tool [27] allows system administrators to run simple, synthetic variations on a user-supplied I/O trace against a simulator, to help visualize the performance consequences of each parameter setting (including RAID levels). Although it assists humans in exploring the search space by hand, it does not automatically search the parameter space itself.

Apart from the HP AutoRAID, none of these systems provide much, if any, assistance with mixed workloads.

The work described here is part of a larger research program at HP Laboratories with the goal of automating the design, construction, and management of storage systems. In the scheme we have developed for this, we run our solver to develop a design for a storage system, then implement that design, monitor it under load, analyze the result, and then re-design the storage system if necessary, to meet changes in workload, available resources, or even simple mis-estimates of the original requirements [4]. Our goal is to do this with no manual intervention at all – we would like the storage system to be completely self-managing. An important part of the solution is the ability to design configurations and data layouts for disk arrays automatically, which is where the work described in this paper contributes.

7 Summary and conclusions

In this paper, we presented a variety of methods for selecting RAID levels, running the gamut from the ones that consider each store in isolation and make irrevocable decisions to the ones that consider all workload interactions and can undo any decision. We then evaluated all schemes for each family in isolation, and then compared the cost of solutions for the best representative from each family. A set of real workload descriptions and models of commercially-available disk arrays was used for the performance study. To the best of our knowledge, this is

Workload	*Taggers*		*Partly adaptive*		*Fully adaptive*	
filesystem	92	(±14)	131	(±42)	273	(±53)
filesystem-lite	51	(±3)	85	(±33)	232	(±28)
oltp	212	(±29)	279	(±46)	669	(±155)
scientific	66	(±5)	116	(±49)	277	(±55)
tpcd30-1x	44	(±10)	85	(±23)	782	(±197)
tpcd30-2x	265	(±49)	393	(±92)	3980	(±1414)
tpcd30-4x	1098	(±159)	2041	(±739)	24011	(±7842)
tpcd300-query-1	689	(±44)	1751	(±2719)	1541	(±300)
tpcd300-query-5	1517	(±85)	2907	(±3593)	4572	(±1097)
tpcd300-query-7	1556	(±90)	2401	(±2126)	5836	(±1345)
tpcd300-query-9	1680	(±73)	2693	(±2362)	6647	(±2012)
tpcd300-query-10	1127	(±77)	2144	(±1746)	2852	(±563)
mean	**700**	**(±633)**	**1252**	**(±2016)**	**4306**	**(±6781)**

Table 5: Mean and (standard deviation) of the CPU time in seconds, for each workload and RAID selection method for the FC-60 array.

the first systematic, automatable attempt to select RAID levels in the published literature.

The simpler tagging schemes are similar to accepted knowledge and to the back-of-the-envelope calculations that system designers currently rely upon. However, they are highly dependent on particular combinations of devices and workloads, and involve hand-picking the right values for many constants, so they are only suitable for limited combinations of workloads and devices. Furthermore, because they put restrictions on the choices the solver can make, they result in poorer solutions.

Integrating RAID level selection into the store-to-device assignment algorithm led to much better results, with the best results being obtained from allowing the solver to revise its RAID-level selection decision at any time.

We showed that the benefits of the fully-adaptive scheme outweigh its additional costs in terms of computation time and complexity. Analysis of the utilization data from the fully-adaptive solver solutions showed that some of the solutions it generated in our experiments were provably of the lowest possible cost (e.g., when the capacity of every disk, or the bandwidth of all but one array, were fully utilized).

For future work, we would like to explore the implications of providing reliability guarantees in addition to performance; we believe that the fully-adaptive schemes would be suitable for this, at the cost of increased running times. We would also like to automatically choose components of different cost for each individual LU within the arrays, e.g., decide between big/slow and small/fast disk drives according to the workload being mapped onto them; and to extend automatic decisions to additional parameters such as LU stripe size and disks used in an LU.

Acknowledgements: We thank Arif Merchant, Susan Spence and Mustafa Uysal for their comments on earlier drafts of the paper.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4), November 2001.

[2] G. A. Alvarez, W. Burkhard, L. Stockmeyer, and F. Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 109–20, June 1998.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical report HPL–SSP–2001–4, Hewlett-Packard Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *File and Storage Technologies Conference (FAST)*, Monterey, January 2002.

[5] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: An ap-

proach to solving the workload and device configuration problem. Technical report HPL–SSP–2001–5, Hewlett-Packard Laboratories, June 2001. http://www.hpl.hp.com/SSP/papers/.

[6] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd – an efficient scheme for tolerating double-disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, February 1995.

[7] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *5th International Workshop on Quality of Service*, Columbia university, new York, NY, June 1997.

[8] P. Chen and E. Lee. Striping in a RAID level 5 disk array. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 136–145, May 1995.

[9] P. Chen and D. Patterson. Maximizing performance in a striped disk array. In *International Symposium on Computer Architecture (ISCA)*, pages 322–331, May 1990.

[10] W. Fernandez de la Vega and G. Lueker. Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[11] L. Hellerstein, G. Gibson, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[12] Hewlett-Packard Company. *HP SureStore E Model 30/FC High Availability Disk Array—User's Guide*, August 1998. Publication A3661–90001.

[13] Hewlett-Packard Company. *HP SureStore E Disk Array FC60—User's guide*, December 2000. Publication A5277–90001.

[14] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *5th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 20, pages 23–35, October 1992.

[15] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, December 1974.

[16] C. Kenyon. Best-fit bin-packing with random order. In *Symposium on Discrete Algorithms*, pages 359–364, January 1996.

[17] E.K. Lee and R.H. Katz. Performance consequences of parity placement in disk arrays. In *4th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–199, Santa Clara, CA, April 1991.

[18] A. Merchant and G. A. Alvarez. Disk array models in Minerva. Technical Report HPL–2001–118, Hewlett-Packard Laboratories, April 2001. http://www.hpl.hp.com/SSP/papers/.

[19] C.-I. Park and T.-Y. Choe. Striping in disk array RM2 enabling the tolerance of double disk failures. In *Supercomputing*, November 1996.

[20] D.A. Patterson, G.A. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD international Conference on the Management of Data*, pages 109–116, Chicago, IL, 1988.

[21] D. Stodolsky, M. Holland, W.V. Courtright II, and G. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–35, August 1994.

[22] P. Triantafillou and C. Faloutsos. Overlay striping and optimal parallel I/O for modern applications. *Parallel Computing*, 24(1):21–43, January 1998.

[23] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*, August 2001.

[24] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–36, February 1996.

[25] John Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the International Workshop on Quality of Service (IWQoS'2001)*, Karlsruhe, Germany, June 2001. Springer-Verlag.

[26] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 146–56, May 1995.

[27] P. Zabback, J. Riegel, and J. Menon. The RAID configuration tool. Technical Report RJ 10055 (90552), IBM Research, November 1996.

Appia: automatic storage area network fabric design

Julie Ward, Michael O'Sullivan†, Troy Shahoumian, and John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA and †*Stanford University, Stanford, CA*
jward@hp.com, mosu@stanford.edu, troy_shahoumian@hp.com, john_wilkes@hp.com

Abstract

Designing a storage area network (SAN) fabric requires devising a set of hubs, switches and links to connect hosts to their storage devices. The network must be capable of simultaneously meeting specified data flow requirements between multiple host-device pairs, and it must do so cost-effectively, since large-scale SAN fabrics can cost millions of dollars. Given that the number of data flows can easily number in the hundreds, simple overprovisioned manual designs are often not attractive: they can cost significantly more than they need to, may not meet the performance needs, may expend valuable resources in the wrong places, and are subject to the usual sources of human error.

Producing SAN fabric designs automatically can address these difficulties, but it is a non-trivial problem: it extends the NP-hard minimum-cost fixed-charge multicommodity network flow problem to include degree constraints, node capacities, node costs, unsplittable flows, and other requirements. Nonetheless, we present here two efficient algorithms for automatic SAN design. We show that these produce cost-effective SAN designs in very reasonable running times, and explore how the two algorithms behave over a range of design problems.

1 Introduction

A SAN (storage area network) connects a group of servers (or hosts) to their shared storage devices (such as disks, disk arrays and tape drives) through an interconnection fabric consisting of hubs, switches and links. We present results for designs using today's dominant SAN fabric for the SCSI block-level protocol, FibreChannel [13]. The storage industry is in the process of adding switched Ethernet as an alternative block-level network transport. We believe that our work applies equally to both, and could also usefully be applied to file-based storage systems, and even general-purpose local-area networks (LANs).

An example FibreChannel SAN is shown in Figure 1.

SANs offer many advantages over direct-connected local storage, including superior connectivity of servers to storage devices, better utilization of storage resources, centralized administration and management, increased scalability, and improved performance. In spite of these advantages, the adoption of SANs has been relatively slow. Some of this is due to interoperability difficulties between vendors, but as these are being resolved, the next barrier appears to be the complexities associated with designing the SANs, because this involves all of the problems of network design – in an environment with essentially no automatic flow control, and zero tolerance for packet loss, due to the low-level nature of the SCSI protocol.

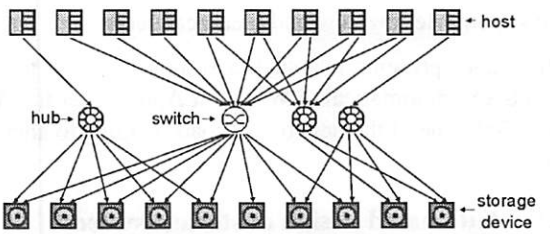


Figure 1: A simple, single-layer SAN fabric. Hosts appear in the top row, devices in the bottom row, and switches and hubs in between.

As a result, designing even small SANs requires considerable time and effort from IT experts. Their manual methods often result in expensive, overprovisioned designs – and this becomes more of a problem as the designs get larger and more complex. This matters: it is not difficult to spend 10–20% of the total storage system cost on the SAN fabric elements, and SAN designs of a scale that require an investment of millions of dollars in the SAN fabric alone are becoming more common. We have witnessed a factor of three difference in the cost of a SAN between a manual design ($4m) that took several days and an automatically-generated one ($1.4m) that took a few minutes.

Design mistakes can be subtle and therefore easy to overlook, yet potentially very costly; poor performance is commonplace, and downtime in failure situations can result if the fault-tolerance aspects are mis-designed. As SANs grow to include hundreds or even thousands of

storage devices, it becomes increasingly difficult, even for SAN experts, to manually design cost-effective and reliable SANs.

We believe that the most effective approach to these problems is to automate the design of SANs. Such designs must take into account the performance demands (to avoid queuing or packet loss), and they should try to minimize system cost, because SAN components are quite expensive. The result would enable the wider deployment of SANs, as well as increase the likelihood that the systems deployed would meet real needs.

This paper presents just such a solution: a tool to design SANs automatically. We call it Appia, after the Appian Way, one of the network of roads leading to ancient Rome.

1.1 Automated design of storage systems

Appia was developed to operate in concert with a set of tools that select and design the storage-device portions of a complete storage system [2, 4]. These tools use workload and device performance information to select and configure storage devices, and then determine appropriate data placements on those devices. Their goal is to design a system that meets performance goals with high reliability at low cost. A side effect is that the tools' output includes information about the workload data-flows from each host to each storage device: precisely the information that is needed to design the SAN fabric to connect the hosts to their storage.

Such tools significantly reduce the human intervention required to design storage systems: people can express their needs at a relatively high level, and the tools can design a storage system to meet their needs, taking into account all the low-level details, such as predicting the complex performance effects that result from mixing workloads on shared storage devices. Better yet, such tools can be used in an automatic control loop, allowing the storage system design to evolve completely automatically when dealing with load and system changes, without the need for human intervention.

We wanted to achieve the same benefits for SAN design. The results presented here are the first outcome of that goal. In particular, we present two algorithms for cost-effective SAN fabric design. These two approaches, which we call *FlowMerge* and *QuickBuilder*, demonstrate complementary strengths. *FlowMerge*, which is more computationally intensive, tends to find lower-cost designs for SANs with sparse connectivity requirements, whereas *QuickBuilder* excels when connectivity requirements are dense. We found that the better of two designs is, on average, within 33% of the optimal design cost for empirical test problems that are small enough to solve optimally. Moreover, these designs are found in a few minutes or less for SANs with 50 hosts and 100 devices, a size typical of the largest current installations. Because of their complementary strengths, both algorithms are included in Appia.

1.2 Structure of the paper

The remainder of this paper is organized as follows. Section 2 presents a statement of the SAN fabric design problem, including notation and related work. Section 3 presents an overview of the *FlowMerge* algorithm for finding cost-effective SAN fabric designs. The *QuickBuilder* algorithm is presented in §4. In §5 we present computational results comparing the effectiveness of the two algorithms. Furthermore, for small problems we compare the cost of designs produced by *FlowMerge* and *QuickBuilder* with the cost of optimal designs. Future work and conclusions are presented in §6 and §7.

2 The SAN design problem

The SAN design problem can be stated quite simply: we are given a set of hosts, a set of storage devices, and a set of requirements in the form of data flows between host-device pairs. Each flow has a desired bandwidth. The goal is to build a minimum-cost SAN to support all of these requirements simultaneously. To do so, one must select a set of fabric nodes (switches and hubs), a set of links connecting pairs of nodes (hosts, devices and fabric nodes), a topology with which to join these together, and a single path through the network for each flow. (The single-path restriction arises from SCSI request-ordering constraints.)

The resulting fabric design must be *feasible* - that is, it must satisfy constraints that ensure it is buildable, and it must support the connection and performance requirements. These constraints are: (1) the number of links connected to a host, device or fabric node must not exceed the number of ports available there (these restrictions are called *degree constraints*) and (2) the flow routing must honor the bandwidth limitations of links and fabric nodes. Because packets travel differently through hubs and switches, their bandwidth constraints differ. Packets routed into a switch are forwarded directly to the next destination in their path. In contrast, packets routed into a hub are transmitted through all connected hubs and all links attached to these hubs; they are seized by their next destination. Thus, the total flow into an interconnected set of hubs is limited by the minimum of the bandwidth of each individual hub, the bandwidth of each connected link, and the bandwidth of each port used by these links. The bandwidth of switches is therefore more efficiently utilized than hub bandwidth.

Data about the flows is readily available from solutions to the storage-system and data-placement design problems [2, 4], but it may also be obtained from the tried and true techniques of measurement of an existing system or estimation. Obviously, no design tool is better than the inputs it is given – but the comparison point here is manual design, not complete knowledge of the system's future behavior. It is easy enough to build in a certain amount of "slack", to allow for errors, or anticipated future growth. Indeed, we believe that it is better to have the slack specified up front as part of the goal, so that the design system can take it into account, rather than trying to build in slack "after the fact" by adding excess SAN elements in places where they may not do the most good.

The design algorithms we describe run fast enough that they can be used in interactive "what if" scenario exploration, in conjunction with manual input from a SAN design expert. The low-cost designs the tools produce may not always "look pretty"; some people prefer greater symmetry in their solutions, even at the expense of greater cost. As such, we believe it is important to use this kind of tool – at least at first – in a context where there is a chance for experts to modify the output it produces. Nonetheless, it is our aim to develop tools that can be placed into a completely automatic design-deploy-monitor-redesign loop.

2.1 Related work

SAN design is currently done manually by IT experts, who use error-prone ad-hoc methods or canned topologies that often result in grossly overprovisioned designs. While overprovisioning can be advantageous, it is important that it is done strategically to provide high performance, scalability, reliability, and robustness to changes in requirements. Some canned designs currently in use, such as the Brocade Core-Edge architecture [9], possess these characteristics. They are used when the SAN designers have no systematic way to predict the connectivity and data flow requirements in their SANs, and so opt for full connectivity between hosts and devices. But this flexibility comes at a very high price: many fabric elements are needed to provide this connectivity, especially at high bandwidths. In general, when any information is available about SAN requirements, far more cost-effective designs can be found.

As part of our search for algorithms to apply to this problem, we turned to the literature on network design. Unfortunately, most traditional network design approaches only address link costs, because switches are cheaper than trenching in wide area telephone networks, which are the target of most of this work. In the SAN case, the reverse is usually the case: in mid-2001, a fully loaded 64-port FibreChannel "storage director" (a high-end fabric switch) costs close to half a million dollars, while individual fibre links for use within a data center are priced around $100–$500. As a result, much of the existing research in network design proved less applicable than we had hoped.

In particular, the SAN fabric design problem generalizes and extends several NP-hard problems in network design. For example, it generalizes the nonbifurcated network loading problem [21, 6, 3, 16, 17]. In this problem, there are several commodities, each with an origin and destination node in the network, and a required amount of the commodity that must travel through the network between these nodes. One must choose a minimum cost set of capacitated links connecting a known set of nodes to satisfy these flow requirements simultaneously. The term "nonbifurcated" refers to the requirement that a single route for each commodity must be selected; i.e., flows cannot be split across multiple paths. Each link has an associated fixed cost, and multiple links between a given pair of nodes may be selected. This problem contains the Steiner tree problem, known to be NP-complete, in which one must find the minimum cost set of links to connect a given subset of the nodes in a network. (See [23] for a survey of work on the Steiner tree problem.) The nonbifurcated network loading problem is NP-hard even when all commodities share a single source [21].

If we relax the constraint that flows cannot be split, the SAN design problem generalizes the multicommodity network design problem [20, 8, 7, 19, 22, 10, 5]. This problem is known to be NP-hard even in the single commodity case [15]. Like the nonbifurcated network loading problem, it involves choosing a set of capacitated, fixed-cost links to connect a set of nodes to satisfy multicommodity flow requirements. Any number of links between a pair of nodes can be selected. In this case, however, flows can be split. Even so, multicommodity network design problems are notoriously difficult to solve in practice. This is true because their integer programming formulations' LP relaxations do not provide tight lower bounds. Even finding feasible solutions is often difficult. Surveys of work in this area are given in [18, 1, 24].

In the NP-hard problems mentioned above, one must find a minimum cost set of links to route the flows, when the nodes in the network are known. The SAN fabric design problem generalizes these problems, in that the nodes in the network are not known *a priori*. One must choose a set of hubs and switches with which to build the interconnection fabric between hosts and devices. Several different types of hubs and switches may be available, differing in attributes such as cost, bandwidth, and number of available ports; an arbitrary number of instances of each type may be used in the SAN. It is possible, however, to

construct a candidate fabric node set containing the optimal set. Few authors have considered network design problems in which the topology is unknown. The Steiner tree problem is a special case of the capacitated network design problem in which some nodes may optionally be excluded from the network. The integer programming formulation of network design problems grows in dimension exponentially with the size of the set of nodes considered, and thus it is essential to find a small candidate node set. Unfortunately in the SAN design context it may be difficult to determine such a candidate set of reasonable size due to the number of different node types considered.

SAN design also generalizes other network design problems by associating capacity and cost with nodes. [17] includes node costs, and [27, 14] consider node capacities. A node's cost and capacity can be handled within the context of standard network design problems at the expense of an additional node and arc: each capacitated or cost-inducing node can be replaced by two uncapacitated and costless nodes with an arc between them possessing the original node's cost and capacity attributes. (This assumes unidirectional links, which will not always be the case in future SAN design problems.)

Another confounding feature of the SAN design problem is the presence of degree constraints on nodes. Degree constraints appear only in special cases of the network design problem such as the degree-constrained minimum spanning tree problem [12, 11, 25], known to be NP-hard [15].

The many features of the SAN design problem have been addressed individually or in small subsets in the work mentioned above. The first to address all of its features in a common framework was [26], in which an algorithm called *Merge* was presented. *Merge* found cost-effective designs for small problems but failed to find feasible designs for larger problems. The algorithms presented here are proven to find feasible designs under a reasonable set of conditions, and their designs are generally more cost-effective.

2.2 Notation

Some notation will be useful in describing our approaches. Let $\mathcal{H}$ and $\mathcal{D}$ represent the sets of hosts and devices, respectively. Denote the set of flows by $\mathcal{F}$. Let $\mathcal{N}$ be the set of all types of switches and hubs available. Each component $i \in \mathcal{H} \cup \mathcal{D} \cup \mathcal{N}$ has a maximum number of ports p_i, each with cost π_i. Although a SAN could be built from several different types of links differing in bandwidth and cost, we restrict attention in this paper to the case when there is one available link type whose bandwidth is β and cost is γ. To simplify exposition, we also assume that all ports have bandwidth β, though ports may differ in cost. Finally, fabric node type $n \in \mathcal{N}$ has cost c_n and maximum aggregate bandwidth b_n. The SAN fabric design problem defined by given sets of hosts, devices, flows and nodes is denoted by P.

3 The *FlowMerge* algorithm

The first of our algorithms is called *FlowMerge*, which earns its name from the way it pulls together separate flows into sets of flows that share fabric nodes. It was inspired by this simple fact: when two flows with a common host or device are routed together through a link, they conserve a port on that host or device. *FlowMerge* attempts to use fabric nodes in a way that alleviates a shortage of host and device ports, by selecting subsets of flows with common hosts or devices to route together through links.

FlowMerge is a recursive algorithm that creates a SAN design by introducing, at each recursive application, a set of fabric nodes and links, with no links between fabric nodes in the set. When the algorithm terminates, the fabric design consists of one or more "layers" of nodes, where there are links between but not within layers. An example of a layered fabric produced by *FlowMerge* is shown in Figure 2. The top and bottom rows of components contain hosts and devices, respectively, and the remaining components are fabric nodes.

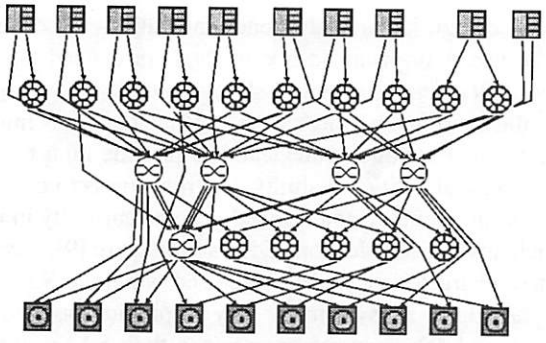


Figure 2: A sample SAN fabric produced by *FlowMerge*

The basic building block of a *FlowMerge* fabric is a *single-layer fabric*. This is a fabric that has no links between fabric nodes, so that each flow requirement is routed either along a direct link between its host and device, or along a two-link path that passes through a single fabric node. Figure 1 depicts an example of a single-layer fabric. In §3.1 we describe the procedure that introduces a single layer of nodes, which we call *Single-Layer FlowMerge*. In §3.2 we outline the recursive procedure that creates a multi-layered fabric through successive calls to *Single-Layer FlowMerge*.

3.1 *Single-Layer FlowMerge*

The input to *Single-Layer FlowMerge* is a set $\mathcal{H}$ of hosts, a set $\mathcal{D}$ of devices, and flow requirements $\mathcal{F}$ between them. *Single-Layer FlowMerge* produces a series of single-layer fabric designs to support the flow requirements. Each design in the series is feasible with respect to all except, possibly, the degree constraints on hosts and devices. The initial design consists of a direct host-device link for each flow. This design is typically infeasible because one or more hosts or devices has fewer ports than incident links. The difference between the numbers of incident links and available ports on a given host or device is called its *port violation*. Each subsequent design in the series has a smaller total port violation than the previous design, or a lower cost than the previous design if both designs are feasible.

To see how this series of designs is obtained, consider an arbitrary single-layer fabric. Associated with each fabric node in the design is a subset of flow requirements routed via that node. Similarly, associated with each direct host-device link in the fabric is a subset of flows routed along that link. In general, the flow requirements are partioned into disjoint subsets, such that each flow requirement is in exactly one subset. Each subset in the partition has an associated fabric node or direct host-device link through which all flows in the subset are routed. We call these subsets *flowsets*.

Single-Layer FlowMerge begins with the finest partition of the flow requirements: each flow is in its own flowset. At each iteration, a new, coarser, partition is obtained by merging two flowsets together. When merging two flowsets, we must select a fabric node type among available types with which to route the flows in the merged flowset, and the links connecting hosts and devices to the node along which we route the flows. The node type is selected based on the number of ports available on the node and the cost of using the node (including the cost of required ports and links). We select the flowsets to merge to alleviate port violations, favoring reductions on the hosts and devices with the most severe violations. Cost is a tie-breaker criterion. Once two flowsets are merged, they are never split. *Single-Layer FlowMerge* continues merging flowsets until either no two flowsets can be merged, or all port violations have been eliminated and no merger produces a cost savings. *Single-Layer FlowMerge* terminates, because after a finite number of mergers (one less than the number of flows) only a single flowset remains, so no further mergers are possible. Figure 3 demonstrates how *Single-Layer FlowMerge* works on a small example.

Pseudocode for the *Single-Layer FlowMerge* algorithm is shown in Figure 4. We use the following notation:

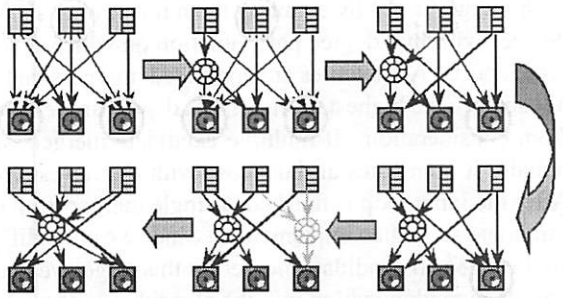


Figure 3: Example application of *Single-Layer FlowMerge*. The problem has 3 hosts and 3 devices, each with 2 ports, and a single type of switch available with 8 ports. The eight flows in the problem each have bandwidth 33 MB/s. Links and ports have bandwidth 100 MB/s. Six successive designs are shown, beginning with the one that assigns each flow to its own link. In each design, hosts and devices with the highest port violation are circled. For example, in the first design, the highest port violation is one: there are two hosts and two devices each with three incident links and only two ports. Each design in the series reduces the port violation on one host or device from the previous design by merging two flowsets together. After four mergers, all port violations are eliminated. The last merge eliminates one fabric node and thereby reduces the cost of the fabric.

- $\mathbf{F}$ is the partition of the set of flows $\mathcal{F}$ into flowsets (more explicitly, $\mathbf{F}$ is a collection $\{J : J \subset \mathcal{F}\}$ with the property that $\cup_{J \in \mathbf{F}} J = \mathcal{F}$ and $K \cap J = \emptyset$ for all $J, K \in \mathbf{F}$, $J \neq K$);
- $\mathcal{N}$ is the set of available fabric node types;
- l is the single available link type;
- $\mathcal{M} \subset \{(J,K,n) : J,K \in \mathbf{F}, J \neq K, n \in \mathcal{N} \cup \{l\}\}$ is a set of triples consisting of two flowsets and one node type or link;
- $score_m$ is a function defined on elements of $\mathcal{M}$.

We refer to elements of $\mathcal{M}$ as *mergers* because they represent the combinations of flowset pairs and node types that are candidates for merging.

In the *Single-Layer FlowMerge* psuedocode, each application of the outer loop results in a merger. We start an application of this loop by initializing the set of candidate mergers $\mathcal{M}$ to be all possible flowset pair-node combinations, and then eliminating infeasible combinations. Next, we compute the port violations on hosts and devices. If there are candidate mergers left to consider, we refine this set in the inner "Else while" loop. This loop considers port violation degree ranging from the current worst, v, down to 1. For each such degree, it "scores"

each merger in $\mathcal{M}$ by counting the number of hosts and devices with that degree port violation on which it conserves ports. After scores are computed, mergers that do not achieve the highest score for this degree are removed from consideration. If multiple candidate mergers still remain, it eliminates all but those with the lowest cost. After the inner loop is finished, a single merger from the candidate set is then implemented. Since we are indifferent between all candidate mergers at this stage, we could introduce randomization into the algorithm in the selection of the merger from the final set of candidates.

Scores computed in the inner loop can be largely reused in successive applications of the outer loop. In our implementation, they are updated for flowset pairs containing hosts or devices whose port violation was reduced in the prior merger.

3.2 *Multi-Layer FlowMerge*

When *Single-Layer FlowMerge* is applied to a SAN fabric design problem, it will reduce at least one host's or device's port violation by at least one. (We omit the details of this proof in the interest of brevity.) However, *Single-Layer FlowMerge* may not successfully eliminate all port violations on hosts and devices. In this case, it is reapplied recursively to generate cascading layers of fabric nodes. Pseudocode for this recursive application, which we call *Multi-Layer FlowMerge*, is shown in Figure 5.

The central idea behind the recursion is as follows. We first apply *Single-Layer FlowMerge* to a SAN fabric design problem P. If all host and device port violations are eliminated from P, we have found a feasible SAN fabric design. At this point, we can stop, since *Single-Layer FlowMerge* found no cost-saving mergers and introducing new fabric nodes would only increase costs.

If instead there are remaining host and/or device port violations, the current set of fabric nodes is insufficient. We address the host port violations first, independently of the device port violations, by recasting the problem as a new SAN fabric design problem P_H that has only host port violations and no device port violations. The hosts of P become hosts of P_H. Subsets of flows in problem P are aggregated together to become flows for problem P_H according to their assignment to links in the one-layer solution to P. More specifically, for each flowset and each link into the flowset's fabric node, a new flow is created in P_H whose bandwidth is the aggregate bandwidth of flows assigned to that link. The new flow's device in P_H is the fabric node itself. If instead its flowset has no fabric node (and thus has a single direct link between a host and device), all flows routed along that link are aggregated into a single flow in P_H. For this flow we create a "dummy" device in P_H with a single port that costs the

Single-Layer FlowMerge
Input: a SAN fabric design problem P.
Output: a set of flowsets $\mathbf{F}$ and a fabric node for each flowset.

Let $\mathbf{F} = \{\{f\} : f \in \mathcal{F}\}$.
While (true){
  Let $\mathcal{M} = \{(J,K,n) : J,K \in \mathbf{F}, J \neq K, n \in \mathcal{N} \cup \{l\}\}$.
  Remove from $\mathcal{M}$ all elements that represent infeasible mergers.
  Compute the port violation on each source and terminal with respect to the current set of flowsets and their associated nodes and link. Let v be the highest port violation among them.
  If $\mathcal{M} = \emptyset$, break.
  Else while $(v > 0)$ and $(|\mathcal{M}| > 1)$ {
    For each $m \in \mathcal{M}$ {
      Let $score_m = 0$.
      For each source and terminal c with port violation v
        If merger m reduces the port violation on c
          Let $score_m = score_m + 1$.
      Remove elements of $\mathcal{M}$ which did not achieve the highest score.
      Let $v = v - 1$.
    }
    For each $m \in \mathcal{M}$
      Compute the cost of merger m.
    Remove mergers in $\mathcal{M}$ which did not achieve the lowest cost.
  }
  Return a random $\hat{m} = (\hat{J},\hat{K},\hat{n}) \in \mathcal{M}$.
  If the merger $\hat{m}$ reduces the port violation on at least one source or terminal with a positive port violation, or if the merger has a negative cost, perform the merger: delete $\hat{J}$ and $\hat{K}$ from $\mathbf{F}$, discarding their respective nodes, and replace with a new flowset $\hat{J} \cup \hat{K}$ with a node of type n.
  Otherwise, break.
}
Return flowsets in $\mathbf{F}$ and their associated fabric nodes.

Figure 4: *Single-Layer FlowMerge*

```
Multi-Layer FlowMerge(P, L)
Input:  a SAN fabric design problem P and a
layer number L.
Output:  a feasible SAN fabric design
consisting of one or more layers of fabric
nodes, with no links between nodes in a given
layer.
   Apply Single-Layer FlowMerge to P.
   If there are remaining host port
   violations in current solution to P {
      Recast problem as new Multi-Layer
      FlowMerge problem P_H.
      Apply Multi-Layer FlowMerge(P_H, L-1).
      Add fabric for P_H to fabric for P.
   }
   If there are remaining device port
   violations in current solution to P {
      Recast problem as new Multi-Layer
      FlowMerge problem P_D.
      Apply Multi-Layer FlowMerge(P_D, L+1).
      Add fabric P_D to fabric for P.
   }
   If there are no remaining port violations
   in P
      Return fabric for P.
```

Figure 5: *Multi-Layer FlowMerge*

same as its original device's ports. Thus, the set of devices in P_H consists of fabric nodes from P and dummy devices corresponding to devices from P; none of these have port violations.

We then apply *Multi-Layer FlowMerge* to the P_H and create a multi-layered fabric for that problem. The next step is to incorporate the fabric for P_H into the solution we are building up for P. P_H's fabric layers are inserted into the fabric of P.

Similarly, if device port violations remain in P after the application of *Single-Layer FlowMerge*, then a new problem P_D is created in a way that mirrors the creation of P_H. It has all devices from P as its devices, aggregated flows from P as its flows, and hosts consisting of fabric nodes and dummy hosts corresponding to hosts in P. P_D is solved and its fabric is incorporated into P's solution.

In this brief overview of *Multi-Layer FlowMerge*, we have omitted many details. For example, there are special precautions taken which ensure that there are no links between hubs in the fabric. While this is not strictly necessary, it is the most efficient way to ensure that hub capacity constraints are honored.

3.3 Correctness and Effectiveness

Although the proof will be omitted here, *FlowMerge* finds a feasible SAN fabric design when the following two conditions hold:

> For each host and device, there exists an assignment of its flows to its ports such that the total bandwidth of flows assigned to a port is at most the port's bandwidth (β). (1)

> There is a switch type available having at least three ports and bandwidth at least β. (2)

Assumption (1) is clearly a necessary condition for the existence of a feasible fabric design. Assumption (2) is not necessary, in general, since a small SAN may require no fabric nodes at all. However, it is not at all restrictive; all real switches possess at least 8 ports and typically many more, and have bandwidth many times that of a link. The two assumptions together are sufficient to ensure that *FlowMerge* finds a feasible fabric design.

While we have no analytical optimality bounds on *FlowMerge* designs, we do have empirical results comparing its designs to those produced by *QuickBuilder* and, for small problems, optimal designs.

Our results indicate that *FlowMerge* is very effective at building one-layer fabrics, which are typically sufficient for problems that either have few hosts and devices and have sparse connectivity requirements between hosts and devices. But for SANs that are so large or whose connectivity requirements are so dense that they require multiple fabric layers, it is less effective than *QuickBuilder*. There are several explanations for these results.

First, the class of fabrics *FlowMerge* generates is more restrictive than those built by *QuickBuilder*. *FlowMerge*'s layered fabric structure, where each layer is built myopically, may exclude more cost-effective fabric designs. In each layer it tries to resolve as many port violations as possible before introducing the next layer. It never considers changing a fabric layer that was created in an earlier application of *Single-Layer FlowMerge*.

Second, because *FlowMerge* only considers pairwise mergers, it can get stuck in locally optimal solutions. To see why, suppose it has found a feasible partition for a layer and is seeking only cost-improving mergers. It will quit if no merger is profitable. In many examples, we have seen that a better solution could have been obtained if mergers of more than two flowsets were considered; this occurs frequently in the multilayered solutions.

Allowing backtracking, or permitting non-cost-improving mergers with some small probability (in the spirit of simulated annealing) are techniques that are likely to improve *FlowMerge*'s performance, particularly on problems requiring multiple layers of fabric. Results from our current implementation of *FlowMerge* will be presented in more detail in §5.

4 The *QuickBuilder* Algorithm

In this section we outline a second, two-phased approach to SAN fabric design, called *QuickBuilder*. It is based on the observation that since flows cannot be split across multiple paths in the network, each flow must be assigned to a single port on its host and device.This matters because the way in which flows are assigned to ports has a large impact on the remainder of the SAN design. A clever assignment creates a partition of the host and device ports into disjoint subsets of ports called *port groups*. The port group of port p is a set of ports that includes p; if q is a port in the port group and a flow assigned to q is also assigned to port r, then r is in the port group. In short, the port group of port p includes p, all ports p must communicate with, all ports they communicate with, etc. In the language of graph theory, port groups are the connected components of a graph in which the nodes are ports, and links connect port pairs with common flows assigned to them. The critical insight was that each port group can be treated as an independent, smaller design problem. In general, the fewer ports in a port group, the less fabric is required to support its flows. Thus, the finer the decomposition, the less costly the fabric. *QuickBuilder* seeks an assignment that results in a fine decomposition.

QuickBuilder first assigns each flow requirement to a single port on its host and a single port on its device (the *port assignment* phase); the flow will later be routed through these ports in the second phase. The assignment obtained in the first phase implies a partition into port groups. Fabric can be built for each port group separately.

The second phase of the algorithm considers each port group created in the port assignment phase separately, and finds a fabric to support the flows assigned to its ports. The fabric associated with a port group is an interconnected set of fabric nodes and links called a *module*, from which we obtain the name *module-building* phase for this part of the algorithm. The two phases are described in more detail in §4.1 and §4.2.

Two examples of *QuickBuilder* designs are shown below. The fabric in Figure 7 was developed by *QuickBuilder* with the same inputs that *FlowMerge* used to find the fabric in Figure 2. For this problem, *QuickBuilder*'s assignment of flows to ports led to two port groups, one of which is very large, containing all but two ports. The fabric contains one direct host-device link, and one very large module with three interconnected switches. Figure 6 is a solution to the SAN design problem for which *FlowMerge* designed the fabric in Figure 1. In this fabric, *QuickBuilder*'s port assignment created five port groups. Two port groups are supported by direct links, two larger port groups are supported by hubs, and the largest is supported by two switches connected to each other. In §5 we compare the *QuickBuilder* and *FlowMerge* solutions in more detail.

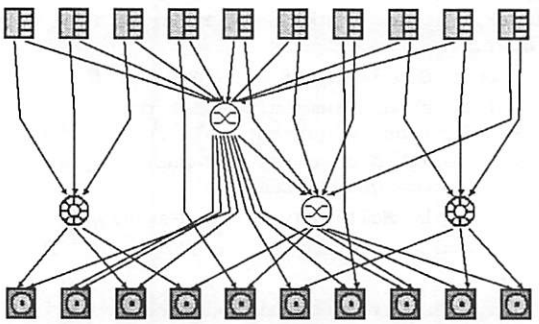


Figure 6: A sample SAN fabric produced by *QuickBuilder* (cf. Figure 1)

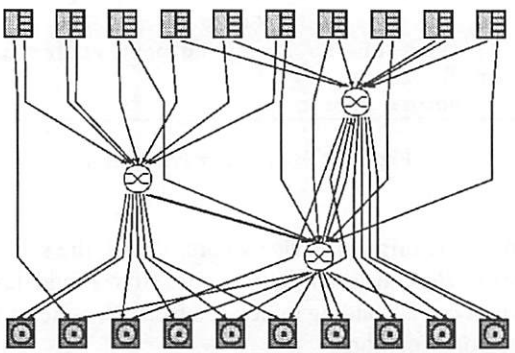


Figure 7: A second *QuickBuilder* SAN fabric (cf. Figure 2)

4.1 Finding port assignments

To find an assignment of flows to host and device ports, *QuickBuilder* considers flows one at a time, looking at each possible combination of host and device ports for each flow's assignment. It chooses the assignment among these that, when added to previous assignments, has the lowest estimated cost. *QuickBuilder* continues making the lowest estimated cost assignment for each flow until all flows have been assigned ports. Although the flows can be assigned in any order, we have found that considering them in order of decreasing bandwidth leads to cost-effective designs.

When estimating the cost of a flow being assigned to particular host and device ports, we account for the previous assignments of flows to these ports. If making this new assignment would cause the total bandwidth of flow to exceed the bandwidth available on either port, then the assignment is infeasible. Furthermore, this port assignment must not preclude the possibility of assigning all of the host's (or device's) unassigned flows to its ports. To determine whether the unassigned flows can be assigned,

we apply an exhaustive bin packing algorithm, where the ports are bins, a port's capacity is the bandwidth unused by previously assigned flows, and the unassigned flows are the items to be packed. If there is no solution, this assignment is infeasible. Infeasible assignments have cost ∞.

If a port assignment is feasible, *QuickBuilder* estimates the cost of supporting the port groups before and after the port assignment is made. The cost of the assignment is the difference between the "after" and "before" cost estimates. The module cost estimation is similar to module construction; we describe both together in §4.2.

4.2 Building modules

The port assignment determined in the first phase of *QuickBuilder* uniquely determines the port groups. In this section, we describe how *QuickBuilder* creates a module to route the flows assigned to ports in a given port group. We also explain how, in the port assignment phase, *QuickBuilder* estimates the module cost for a given port group. For most port groups, the two processes involve the same computations.

When building a module or estimating the cost of a module for a port group, we assume for simplicity a single type of hub h and a single type of switch s to use in the module. Recall that bandwidth, number of ports, and cost of a type n fabric node are b_n, p_n and c_n, respectively. The module-building phase of the algorithm relies upon the assumption that there is a hierarchy among fabric elements, namely, $b_s > b_h$ and $c_s > c_h$. The building and estimation processes depend on properties of the port group. In particular, three cases are considered:

- **Case 1: Using a direct link.** If the port group has only two ports, then a module consisting of a single direct link between the two ports is sufficient. The cost of such a module is simply the cost γ of a link plus the costs of the host and device ports. The estimate of the module building cost is exact in this case.

- **Case 2: Using a (multi-)hub.** If the total flow bandwidth through the ports in the port group is less than b_h, then we use a hub or a *multi-hub*, which is a series of hubs, each connected to the next by a single link. The number of ports available on a hub is p_h; a multi-hub consisting of $i \geq 1$ hubs has $i(p_h - 2) + 2$ available ports. If the number of ports in a port group is k then $H = \lceil (k-2)/(p_h - 2) \rceil$ hubs are required. The module cost is the sum of the cost of the hubs Hc_h, the cost $(H-1)(2\pi_h + c_L)$ of connecting the hubs via links and hub ports, the cost $k(\pi_h + c_L)$ of connecting the host and device ports to the hubs including link and port costs, and the cost of the host and device ports in the port group. In this case, the *QuickBuilder* module cost estimate is also identical to the true cost of a module that would be built for this port group.

- **Case 3: Using a switch module.** If neither of the above conditions holds, then the module must contain at least one switch. We refer to a module that contains one or more switches as a *switch module*. To estimate the cost of a switch module, *QuickBuilder* estimates the number of switches that are needed to support the flows in the port group. In the interest of efficiency, we estimate this number of required switches without determining their exact connectivity in the module and how flows would be routed through them. To do so, we first make the simplifying assumption that all flows in the port group are routed through a single switch of infinite bandwidth. This helps us ignore the effects of flows traveling between multiple switches. *QuickBuilder* calculates the minimum number of ports that would be required to route the port group's flows through this infinite bandwidth switch, and then finds the minimum number of real switches that are required to provide that number of ports. Because there may, in fact, be multiple fabric nodes in the module and flows may travel between them, some of each node's bandwidth will be effectively "wasted" by this inter-node travel.

  To reduce the adverse effect of the infinite-switch assumption upon the estimate, *QuickBuilder* scales up the flows when making this calculation. For each port in the port group, it (temporarily) increases the bandwidth of each flow on that port uniformly by a fixed percentage (typically 10%) or until the capacity of the port is reached. Then, in order of decreasing total assigned flow bandwidth, ports in the port group are "connected" to the first switch port with enough remaining bandwidth to carry that port's flow. If multiple ports in the port group are connected to the same switch port, these ports are instead connected to the smallest required multi-hub which then is connected to the switch port. If k is the number of switch ports used, then $S = \lceil k/p_s \rceil$ is the minimum number of switches of type s needed to provide k ports. The estimated module cost is then the sum of the cost of the switches Sc_s, and the cost of links, hubs and ports used to connect the host and device ports to switch ports.

4.3 Building a switch module

When we are actually building a switch module for a port group, a more elaborate procedure than that of §4.2 is required to determine the exact connectivity of fabric nodes within the switch module. This procedure is outlined here. Switch module construction is a recursive procedure that introduces a series of switches in succession until all flows in the port group can be supported. Its input is a set of *external ports*, each with a set of flows entering the port (called *in-flows*) and exiting the port (called *out-flows*). In the initial call, the external ports are host ports with only out-flows and device ports with only in-flows. On subsequent calls to the procedure, some of the external ports are ports on switches *QuickBuilder* has already added to the switch module. Such ports may have both in- and out-flows.

When building a switch module, *QuickBuilder* first adds a new switch and connects external ports to this switch, selecting the best port to connect according to a merit function. When it connects an external port to the switch, it routes all out-flows (respectively, in-flows) from the port into (out of) the switch. Doing so creates *hanging flows*. These are flows that enter (respectively, exit) the switch via the connection of an external port to a switch port, but have not been assigned to a port by which to exit (enter) the switch. After an external port is connected to a switch port, any resultant hanging flows must be assigned to open switch ports. *QuickBuilder* continues connecting external ports to the switch until the switch is *saturated*, i.e., its bandwidth or port supply has been exhausted. When this occurs, *QuickBuilder* resets the set of external ports to be the current open external ports and switch ports that now contain in-flows and/or out-flows. It then invokes the switch-module-building procedure again on the new external ports.

4.4 Correctness and effectiveness

Like *FlowMerge*, *QuickBuilder* finds a feasible SAN fabric design when conditions (1) and (2) hold. The details of the proof are omitted here.

As with *FlowMerge*, we have no analytical bounds on the cost-effectiveness of *QuickBuilder* designs. However, empirical results indicate that *QuickBuilder* excels at solving large SAN design problems and those with dense flow requirements. In such problems, the flow assignment often results in one port group containing most or all of the ports. Thus, such problems require a large fabric through which almost all host and device ports are interconnected. *QuickBuilder* invokes its switch-module building routine to find this fabric. This routine generally makes very cost-effective use of switches for large port groups by minimizing the bandwidth "wasted" by flows routed through multiple switches in the module.

5 Evaluation of the algorithms

This section summarizes the results we obtained in applying integer programming, *FlowMerge* and *QuickBuilder* to several SAN fabric design problems.

The true test of our algorithms will happen only when their designs are implemented in a real business context and compared to those created manually by experts in the field. This comparison should include several metrics, including cost, performance, availability, scalability and even aesthetics. So far, we have only had a few opportunities to compare our designs to manual ones. Appia found much cheaper designs in these cases. In one notable example, a consultant worked for several days to produce a $4 million design on a problem that Appia solved for $1.4 million in a few minutes. (The consultant used several expensive, 64-port switches, and a completely symmetrical solution; Appia's *FlowMerge* found ways to achieve the same goals using much cheaper 16-port switches.) Nonetheless, we are loath to make strong claims about the benefits of our approach until we have had more opportunities to evaluate it on a wider range of real-world problems.

Nonetheless, our need to design and test Appia required us to generate a wide range of "realistic" test cases where we attempted to introduce elements of the real-world problems we had seen. We sought input from SAN designers in choosing our suite of test problems.

To that end, we generated 240 test problems in 24 categories, each with 10 test problems. The problem categories differed in size (defined by number of hosts and devices), a property we called *port saturation*, and characteristics of a problem feature called the *flow incidence matrix*. Since SANs are currently being designed on many scales, ranging from a handful to a few hundred servers and storage devices, we selected four size categories in this range. A host's or device's port saturation is defined to be its total bandwidth of associated flow requirements divided by the total bandwidth of its ports. The flow incidence matrix is a matrix whose rows correspond to hosts and whose columns correspond to devices. An entry in the matrix equals k if its corresponding host and device have k flows between them. We say a problem is *sparse* if its flow incidence matrix has relatively few positive entries scattered more or less uniformly throughout the rows and columns. Similarly, dense problems correspond to relatively dense and uniform matrices. A *clustery* flow incidence matrix is less uniform, corresponding to the situation when the hosts and devices can be partitioned into "clusters" that contain most of the flow requirements.

5.1 Effectiveness

Table 1 summarizes the computational results for test problems in each category for each of four methods. The first of these methods is the integer program (IP). This approach can solve only the smallest problems, but it does so optimally. A SAN fabric design problem with 10 hosts and 10 devices has over forty thousand binary variables and seventy five thousand constraints, a size far beyond the capabilities of today's commercial IP solvers. The LP (linear programming) relaxation of the IP can be solved for somewhat larger problems; its results are also presented in the chart. The LP relaxation is created by relaxing integrality constraints in the IP. It does not produce usable designs, but it provides a lower bound on the optimal design cost because it solves a less constrained problem. It can therefore be a useful benchmark for heuristics when the optimal cost is not known. We found, however, that the LP bound is quite weak – less than 35% of the optimal cost – for problems with high port saturation.

Statistics for running times of the respective approaches are also given for each category. Notice that in some of the categories, the IP and LP runs were terminated after 24 hours, before solutions had been found.

Results in Table 1 indicate that, on average, *FlowMerge* produces lower cost designs than *QuickBuilder* for smaller problems, whereas for large problems, *QuickBuilder* finds dramatically cheaper designs. The other problem characteristics do not conclusively predict which algorithm is preferable. Figure 8 makes this comparison of *FlowMerge* and *QuickBuilder* design costs more explicit for all but the smallest problems.

The relative advantages of *QuickBuilder* can be seen by a direct comparison of the fabrics in Figure 2 and Figure 7, found by *FlowMerge* and *QuickBuilder* respectively, for the same problem. The *FlowMerge* fabric uses five switches (the darker fabric nodes) and fifteen hubs, and costs $265,080. *QuickBuilder* produced a $133,440 fabric using only three switches. This problem has a very dense requirements matrix and high port saturation. In very dense problems, often every possible assignment of flows to ports results in one port group containing all or most of the host and device ports (to borrow terminology from *QuickBuilder*.) Thus, such problems require a large fabric through which almost all host and device ports are interconnected.

FlowMerge usually needs multiple fabric layers to connect all ports in a large port group. Its myopia in building independent layers and in performing only pairwise mergers impairs its effectiveness in such problems, as discussed in §3.3. For example, in Figure 2, the middle layer of fabric was introduced first without regard for how it would affect future layers, and subsequent layers were built independently of the others. Moreover, it overlooked cost-saving multi-flowset mergers in the outermost layers.

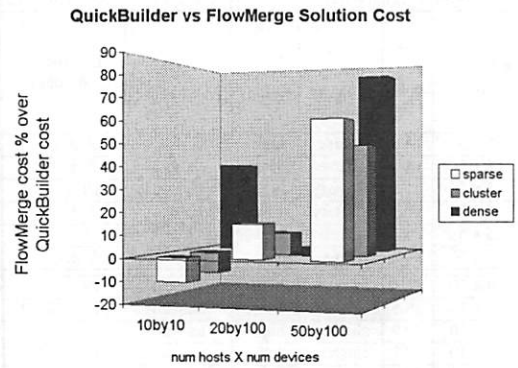


Figure 8: The percentage by which *FlowMerge* design costs exceed *QuickBuilder* design costs, averaged over twenty tests in each of nine categories. The categories are the combinations of the three largest problem sizes (number of hosts and devices each greater than five) and the three flow incidence matrix properties. The horizontal axis indicates the number of hosts and number of devices in the problem category. The bar shade and z-axis position indicate whether the flow requirements matrix is sparse, clustery or dense for that category.

Contrastingly, *FlowMerge*'s relative strengths for less dense problems are apparent when comparing the fabrics in Figure 1 and Figure 6 for the same problem. *FlowMerge*'s $63,720 fabric uses only one switch and three hubs, whereas *QuickBuilder* produced a more expensive $97,120 fabric. This might be explained by *QuickBuilder*'s quite myopic port assignment method, which ignores the flows that have yet to be assigned while making its current assignment. The port assignment determines the decomposition of ports into port groups, and thereby a decomposition of flows into disjoint subsets that can be routed through independent fabric elements. In this particular example, *FlowMerge*'s fabric has four distinct port groups, the largest containing sixteen ports. *QuickBuilder* created five port groups with twenty-four ports in the largest group. Large port groups typically lead to more fabric. *FlowMerge* excels at finding finer decompositions in less dense problems. Thus, *FlowMerge*'s strength is assigning flows to ports in such a way to yield smaller port groups, whereas *QuickBuilder* is better at building modules for large port groups when they are unavoidable. This supports an obvious strategy: run both algorithms, and pick the better solution.

Figure 9 and Figure 10 focus more closely on the smallest problems, with five hosts and five devices, because these problems can be solved optimally by the IP. The

Problem characteristics				Average (in $1000) and standard deviation of fabric cost				Average (in seconds) and standard deviation of solution time			
# hosts x # devices	avg # flows	flow matrix property	port saturation	optimal cost (± std dev)	LP cost (± std dev)	*FlowMerge* cost (± std dev)	*QuickBuilder* cost (± std dev)	optimal time (± std dev)	LP time (± std dev)	*FlowMerge* time (± std dev)	*QuickBuilder* time (± std dev)
5 x 5	14.3	sparse	high	27 ± 13	9 ± 1	31 ± 13	33 ± 12	9,080 ± 13,200	1.4 ± 1.26	0.1 ± 0.00	0.1 ± 0.00
5 x 5	14.0	sparse	low	11 ± 2	8 ± 0	13 ± 3	15 ± 3	1 ± 1	0.0 ± 0.02	0.1 ± 0.04	0.1 ± 0.00
5 x 5	21.8	clustery	high	40 ± 2	10 ± 1	41 ± 2	42 ± 3	19,800 ± 42,500	5.2 ± 1.39	0.3 ± 0.04	0.1 ± 0.00
5 x 5	21.5	clustery	low	12 ± 2	8 ± 0	17 ± 7	18 ± 9	3 ± 3	0.1 ± 0.02	0.3 ± 0.05	0.1 ± 0.00
5 x 5	24.2	dense	high	43 ± 1	11 ± 1	45 ± 1	46 ± 2	3,760 ± 2,370	9.9 ± 2.20	0.4 ± 0.05	0.1 ± 0.00
5 x 5	24.2	dense	low	16 ± 2	9 ± 0	38 ± 0	46 ± 2	1,690 ± 1,690	6.2 ± 0.97	0.4 ± 0.05	0.1 ± 0.00
10 x 10	28.1	sparse	high	unavail	19 ± 1	66 ± 13	77 ± 13	> 24 h	68.6 ± 61.1	0.1 ± 0.00	0.2 ± 0.00
10 x 10	28.5	sparse	low	unavail	17 ± 0	29 ± 3	32 ± 3	> 24 h	1.3 ± 0.83	0.1 ± 0.00	0.2 ± 0.00
10 x 10	40.9	clustery	high	unavail	21 ± 1	88 ± 20	94 ± 17	> 24 h	569 ± 576	0.1 ± 0.05	0.3 ± 0.03
10 x 10	39.7	clustery	low	unavail	17 ± 1	45 ± 11	55 ± 19	> 24 h	36.1 ± 33.5	0.1 ± 0.04	0.3 ± 0.03
10 x 10	94.1	dense	high	unavail	unavail	264 ± 16	138 ± 12	> 24 h	> 24 h	1.7 ± 0.15	0.6 ± 0.05
10 x 10	90.5	dense	low	unavail	unavail	89 ± 9	99 ± 10	> 24 h	> 24 h	1.3 ± 0.14	0.5 ± 0.00
20 x 100	180	sparse	high	unavail	unavail	467 ± 34	478 ± 39	> 24 h	> 24 h	7.5 ± 0.95	2.3 ± 0.17
20 x 100	161	sparse	low	unavail	unavail	342 ± 44	261 ± 29	> 24 h	> 24 h	6.2 ± 0.95	2.1 ± 0.36
20 x 100	226	clustery	high	unavail	unavail	503 ± 36	512 ± 61	> 24 h	> 24 h	15.1 ± 1.67	2.9 ± 0.15
20 x 100	217	clustery	low	unavail	unavail	375 ± 34	322 ± 57	> 24 h	> 24 h	12.7 ± 1.67	2.9 ± 0.32
20 x 100	214	dense	high	unavail	unavail	455 ± 58	441 ± 33	> 24 h	> 24 h	12.2 ± 1.22	2.7 ± 0.15
20 x 100	204	dense	low	unavail	unavail	246 ± 35	278 ± 42	> 24 h	> 24 h	9.5 ± 1.22	2.5 ± 0.16
50 x 100	448	sparse	high	unavail	unavail	1640 ± 231	1030 ± 20	> 24 h	> 24 h	258 ± 7.44	14.4 ± 0.16
50 x 100	402	sparse	low	unavail	unavail	1150 ± 59	718 ± 40	> 24 h	> 24 h	248 ± 6.51	21.9 ± 1.46
50 x 100	607	clustery	high	unavail	unavail	1560 ± 54	1010 ± 22	> 24 h	> 24 h	497 ± 29.4	19.6 ± 0.85
50 x 100	599	clustery	low	unavail	unavail	1010 ± 34	703 ± 32	> 24 h	> 24 h	487 ± 29.4	29.7 ± 2.90
50 x 100	539	dense	high	unavail	unavail	1900 ± 47	1080 ± 29	> 24 h	> 24 h	469 ± 20.7	18.3 ± 0.79
50 x 100	514	dense	low	unavail	unavail	1530 ± 45	805 ± 65	> 24 h	> 24 h	457 ± 20.7	33.2 ± 2.70

Table 1: Summary of computational results for four Appia design methods.
The results in each row are averaged across ten randomly-generated problems of the type shown under "problem characteristics." The first column indicates one of the four problem sizes used: 5 hosts, 5 devices; 10 hosts, 10 devices; 20 hosts, 100 devices; and 50 hosts, 100 devices. The second column reports the average number of flow requirements among problems in the category. The third column indicates qualititative properties of the flow incidence matrix. The fourth column describes the degree of port saturation on hosts and devices. For the "high" port saturation tests, 90% of port bandwidth of the hosts and devices is used, whereas for "low" saturation, only 40% of the port bandwidth is used.
The next four columns provide the average and standard deviation over the category tests of the cost of fabrics found by the four methods. The labels "optimal," and "LP," correspond respectively to the integer program and its LP relaxation. The term "unavail" means that we were unable to compute a result in less that 24 hours for tests in that category.
The last four columns contain the average and standard deviation of the solution times in each category, measured on an HP 9000 model with a PA8600 processor and 4GB of memory, running HP-UX 11.0. Numbers have been rounded to three significant digits.

former shows the relationship between the optimal design cost and the cost of the designs produced by the two heuristics. It indicates that for small problems, *FlowMerge* and *QuickBuilder* find solutions that are, on average, 38% and 55% over the optimal fabric cost, respectively. The fourth bar contains the cost produced by the linear programming (LP) relaxation of the integer program, a lower bound on the optimal cost. In these small problems, the lower cost bound is, on average, half of the optimal cost.

Figure 10 contrasts the fabric costs for individual small tests for each of the four methods. In all of these small tests except for those that have both dense flow incidence matrices and low-saturated ports, *FlowMerge* and *QuickBuilder* find designs that average within 13% and 25% of the optimal design cost, respectively. The heuristics perform less well in the dense and low-saturation cases. The optimal fabrics use only inexpensive hubs, whereas both heuristics use a $24,000 switch, and many expensive switch ports, for each of these problems.

5.2 Efficiency

The graphs in Figure 11 show the algorithms' running times for all 240 test problems as a function of the number of flows in the problem. We chose number of flows as the independent variable because it is the most significant factor in the running times of the two algorithms. For the largest tests, with 50 hosts, 100 devices and 600 flows, *FlowMerge* finds a design in less than 10 minutes, and *QuickBuilder* finds one in less than 40 seconds. This means that adding a *QuickBuilder* run to a *FlowMerge* run is very cheap. Given the target use, it may make sense to use *QuickBuilder* interactively, and then invoke *FlowMerge* in batch mode for final review.

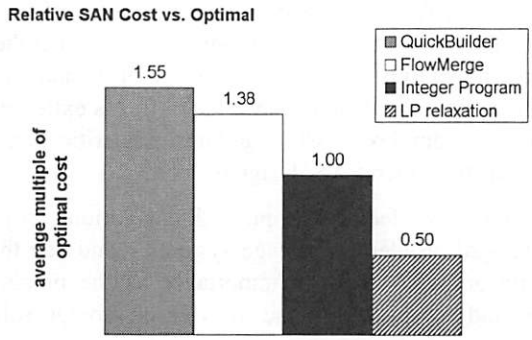


Figure 9: Cost comparisons of the resulting SAN designs for four different design algorithms, averaged across all 5 host, 5 device tests. "Integer program" (IP) produces optimal solutions; *FlowMerge* and *QuickBuilder* are heuristics that produce feasible solutions; LP relaxation produces a guaranteed lower bound, but (in general) infeasible solutions.

6 Future Work

We are actively pursuing several directions of future work:

- Extending the design tools to accommodate high availability requirements. A trivial solution often used for simple SANs is to replicate a single SAN fabric design, but this can become prohibitively expensive when port-count restrictions occur.
- Developing refinements that allow Appia to modify an existing design, rather than design one from scratch. This has obvious practical applications where an existing SAN is being extended; it also introduces some interesting tensions between the desire to produce a high-quality solution, and the desire to minimize the amount of rewiring required on the existing system while trying to use as many of its components as possible.
- Exploring the design of solutions that provide "slack," to allow graceful growth.
- Exploring topology-constrained solutions, such as Brocade's Core-Edge architecture, as one approach to producing designs that may be easier for people to modify by hand at a later date. This is a trivial problem for Appia compared to designing the topology itself – but its existing infrastructure makes it easy to supply this solution for people who prefer it.
- Packaging the tools so that they can be made more widely available, including integrating them more

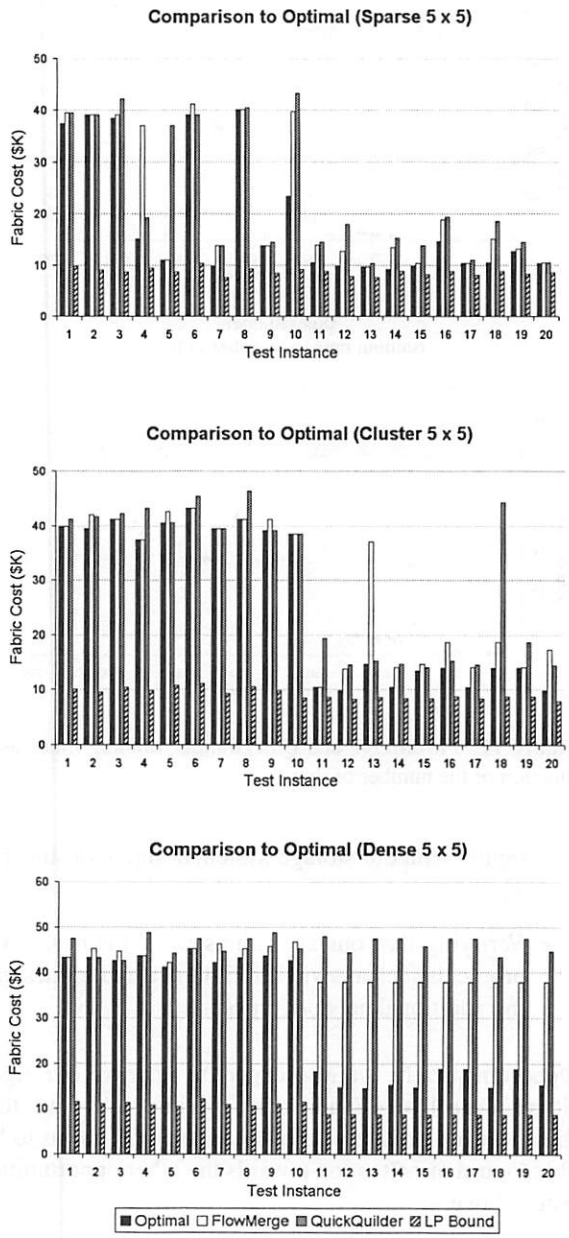


Figure 10: The graphs illustrate SAN solution costs for the four different Appia design algorithms across 20 different problems of the indicated type. For each graph, test instances 1–10 correspond to tests with high port saturation, and tests 11–20 have low-saturated ports. The problem scale was restricted to five hosts and five devices, to allow the optimal (integer programming) algorithm to complete in a reasonable time.

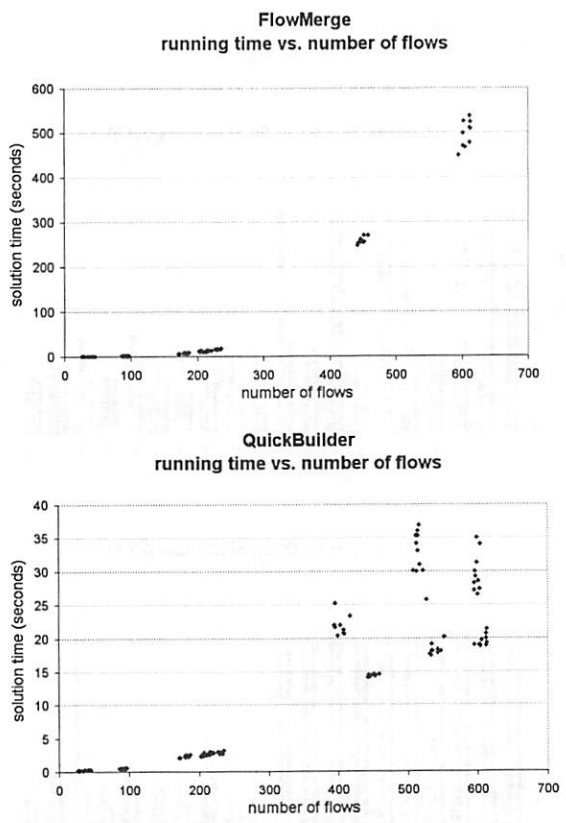


Figure 11: *FlowMerge* and *QuickBuilder* running times as a function of the number of flows.

tightly with our storage-system design tool suite [2, 4].

- Verifying that our algorithms work for new SAN protocols such as switched Ethernet, and for designing additional network types, such as LANs.

Opportunistically, we also expect to improve our algorithms' effectiveness and their runtime – but we feel that these are probably both "good enough" for us to be able to turn our attention towards the other opportunities listed above.

7 Conclusions

The Appia tool and its algorithms are able to design high quality SAN designs. Those designs are quite close to the optimal ones, in cases where we can evaluate them directly – and are several times less expensive than some manual designs we have seen, where over-provisioning by a factor of three "just to be safe" is a common approach.

In our interactions with customers and the domain experts who support them, we have learned that although it is helpful for Appia's SAN designs to be as cost effective as possible, it is probably even more important that they can be shown to be correct – the chance of human error has been greatly reduced. The value of this is extremely high in the complex, mission- and business-critical environments for which SAN design is done.

In summary, we feel that Appia and its algorithms solve a key, hard problem in storage systems – and one that is only going to grow in importance as the number, scale, and complexity of the SAN-based storage solutions grows.

7.1 Acknowledgements

We are endebted to Li-Shiuan Peh for getting the Appia project off the ground and aptly naming it, to Ren Wu and Eric Anderson for their help with implementation of our algorithms, to Rick Rosenthal for his helpful input for the exposition of this paper, and to several other colleagues at HP Labs for their support during this project.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, and M.R. Reddy, *Applications of network optimization*, Network Models (M. O. Ball, T. L. Magnanti, C. L. Monma, and Nemhauser G. L., eds.), Handbooks in Operations Research and Management Science, vol. 7, North Holland, 1995, pp. 1–83.

[2] G.A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A Veitch, and J. Wilkes, *Minerva: an automated resource provisioning tool for large-scale storage systems*, ACM Transactions on Computer Systems (2001).

[3] A. Amiri, *A system for the design of packet-switched communication networks with economic tradeoffs*, Computer Communications **21** (1998), 1670–1680.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, *Hippodrome: running circels around storage administration*, File and Storage technologies (FAST) (Monterey, CA), January 2002.

[5] A. Atamtürk, *On capacitated network design cut-set polyhedra*, Research report, IEOR Department, University of California at Berkeley, December 2000, Available at http://ieor.berkeley.edu/~atamturk.

[6] F. Barahona, *Network design using cut inequalities*, SIAM Journal on Optimization **6** (1996), 823–837.

[7] D. Bienstock, S. Chopra, and O. Gunluk, *Minimum cost capacity installation for multicommodity network flows*, Mathematical Programming **81** (1998), no. 2–1, 177–199.

[8] D. Bienstock and O. Gunluk, *Capacitated network design. polyhedral structure and computation.*, INFORMS Journal on Computing **8** (1996), 243–259.

[9] *Designing next-generation fabrics with Brocade switches*, White paper, Brocade Networks, San Jose, CA,

October 2001, http://www.brocade.com/SAN/pdf/CoreEdgeRev10901.pdf.

[10] S. Chopra, I. Gilboa, and S. T. Sastry, *Source sink flows with capacity installation in batches*, Discrete Applied Mathematics **85** (1998), 165–192.

[11] C.-H. Chu, G. Premkumar, C. Chou, and J. Sun, *Dynamic degree constrained network design: a genetic algorithm approach*, Proceedings GECCO-99. Genetic and Evolutionary Computation Conference. Eighth International Conference on Genetic Algorithms (ICGA-99) and the Fourth Annual Genetic Programming Conference (GP-99), vol. 1, 1999, pp. 141–148.

[12] N. Deo and S.L. Hakimi, *The shortest generalized Hamiltonian tree*, Proceedings of the 6th Annual Allerton Conference, 1968, pp. 879–888.

[13] *Overview of fibre channel technology*, Fibre Channel Industry Association, http://www.fibrechannel.com/technology/, accessed November 2001.

[14] P.C. Fetterolf and G. Anandalingam, *A Lagrangian relaxation technique for optimizing interconnection of local area networks*, Operations Research **40** (1992), 678–688.

[15] M.R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W. H Freeman and Company, San Francisco, CA, 1979.

[16] B. Gavish, *A general model for the topological design of computer networks*, IEEE Global Telecommunications Conference, no. 3, December 1986, pp. 1584–1588.

[17] ———, *Topological design of computer communication networks-the overall design problem*, European Journal of Operational Research **58** (1992), 149–172.

[18] B. Gendron, T. Crainic, and A. Frangioni, *Multicommodity capacitated network design*, Telecommunications Network Planning (B. Sanso and P. Soriano, eds.), Kluwer Academic Press, 1999, pp. 1–19.

[19] J.W. Herrmann, G. Ioannou, I. Minis, and J.M. Proth, *A dual ascent approach to the fixed-charge capacitated network design problem*, European Journal of Operational Research **95** (1996), 476–490.

[20] K. Holmberg and Di Yuan, *A Lagrangean heuristic based branch-and-bound method for the capacitated network design problem*, Proceedings of International Symposium on Operations Research, September 1996, pp. 78–83.

[21] J. M Kleinburg, *Single source unsplittable flow*, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, 1996, pp. 68–77.

[22] T. L. Magnanti, P. Mirchandani, and R. Vachani, *Modeling and solving the capacitated network loading problem*, Operations Research **43** (1995), 142–157.

[23] T. L. Magnanti and L. A. Wolsey, *Optimal trees*, Network Models (M. O. Ball, T. L. Magnanti, C. L. Monma, and Nemhauser G. L., eds.), Handbooks in Operations Research and Management Science, vol. 7, North Holland, 1995, pp. 503–615.

[24] T. L. Magnanti and R.T. Wong, *Network design and transportation planning*, Transportation Science **8** (1984), 1–55.

[25] S.C. Narula and C.A. Ho, *Degree-constrained minimum spanning tree*, Computers and Operations Research **7** (1980), 239–49.

[26] Li-Shiuan Peh, *The appia topology solver: implementation*, Technical report HPL–SSP–98–13, Hewlett-Packard Laboratories, Palo Alto, CA, September 1998, http://www.hpl.hp.com/SSP/papers/#Appia.

[27] V. Sridhar and J.S. Park, *Benders-and-cut algorithm for fixed-charge capacitated network design problem*, European Journal of Operational Research **125** (2000), 622–32.

Aqueduct: Online Data Migration with Performance Guarantees

Chenyang Lu
Department of Computer Science
University of Virginia
chenyang@cs.virginia.edu

Guillermo A. Alvarez John Wilkes
Storage and Content Distribution Department
Hewlett-Packard Laboratories
{galvarez, wilkes}@hpl.hp.com

Abstract

Modern computer systems are expected to be up continuously: even planned downtime to accomplish system reconfiguration is becoming unacceptable, so more and more changes are having to be made to "live" systems that are running production workloads. One of those changes is data migration: moving data from one storage device to another for load balancing, system expansion, failure recovery, or a myriad of other reasons. Traditional methods for achieving this either require application down-time, or severely impact the performance of foreground applications – neither a good outcome when performance predictability is almost as important as raw speed. Our solution to this problem, Aqueduct, uses a control-theoretical approach to statistically guarantee a bound on the amount of impact on foreground work during a data migration, while still accomplishing the data migration in as short a time as possible. The result is better quality of service for the end users, less stress for the system administrators, and systems that can be adapted more readily to meet changing demands.

1. Introduction

Current enterprise computing systems store tens of terabytes of active, online data in dozens to hundreds of disk arrays, interconnected by storage area networks (*SANs*) such as Fibre Channel [4] or Gigabit Ethernet [1]. Keeping such systems operating in the face of changing access patterns (whether gradual, seasonal, or unforeseen), new applications, equipment failures, new resources, and the needs to balance loads to achieve acceptable performance requires that data be moved, or *migrated*, between storage system components – sometimes on short notice. (We note in passing that creating and restoring online backups can be viewed as a particular case of data migration in which the source copy is not erased.)

Existing approaches to data migration either take the data offline while it is moved, or allow the I/O resource consumption engendered by the migration process itself to interfere with foreground application accesses and slow them down – sometimes to unacceptable levels. The former is clearly undesirable in today's global, always-on Internet environment, where people from around the globe are accessing data day and night. The latter is almost as bad, given that the predictability of information-access applications is almost as much a prerequisite for the success of a modern business as is their raw performance. We believe there is a better way; this paper explores our approach to the problem of performing general, online data migrations while maintaining performance guarantees for foreground loads.

1.1. Problem formulation

We formalize the problem as follows. The data to be migrated is accessed by *client applications* that continue to execute in the foreground in parallel with the migration. The inputs to the migration engine are (1) a *migration plan*, a sequence of data moves to rearrange the data placement on the system from an initial state to the desired final state [3], and (2) client application quality-of-service (QoS) demands – I/O performance specifications that must be met while migration takes place. Highly variable service times in storage systems (e.g., due to unpredictable positioning delays, caching, and I/O request reordering) and workload fluctuations on arbitrary time scales [10] make it difficult to provide absolute guarantees, so statistical guarantees are preferable unless gross over-provisioning can be tolerated.

The data migration problem is to complete the data migration in the shortest possible time that is compatible with maintaining the QoS goals.

1.2. QoS contracts

One of the keys to the problem is a useful formalization of the QoS goals. We use the following. A *store* is a logically contiguous array of bytes, such as a database table or a file system; its size is typically measured in gigabytes. Stores are accessed by *streams*, which represent I/O access patterns; each store may have one or multiple streams. The granularity of a stream is somewhat at the whim of the definer, but usually corresponds to some recognizable entity such as an application.

Global QoS guarantees bound the aggregate performance of I/Os from all client applications in the system, but do not guarantee the performance of any individual store or application. They are seldom sufficient for realistic application mixes, for access demands on different stores may be significantly different during migration (as shown in Sections 5 and 6). On the other hand, *stream-level guarantees* have the opposite difficulty: they can proliferate without bound, and so run the risk of scaling poorly due to management overhead.

An intermediate level, and the one adopted by Aqueduct, is to provide *store-level guarantees*. (In practice, this has similar effects to stream-level guarantees for our real-life workloads because the data-gathering system we use to generate workload characterizations creates one stream for each store by default.) Let the *average latency* AL_i of a store i in the workload be the average latency of I/Os directed to store i by client applications throughout the execution of a migration plan, and let the *latency contract* for store i be denoted LC_i. The latency contract is expressed as a *bounded average latency:* it requires that $AL_i \leq LC_i$ for every store i.

In practice, such QoS contract specifications may be derived from application requirements (e.g., based on the timing constraints and buffer size of a media-streaming server), or specified by hand, or empirically derived from workload monitoring and measurements.

We also monitor how often the latency bounds are violated over shorter time intervals than the entire migration, by dividing the migration into equal-sized *sampling periods*, each of duration W. Let M be the number of such periods needed for a given migration. Let the *sampled latency* $L_i(k)$ of store i be its the average latency in the k^{th} sampling period, which covers the time interval $((k-1)W, kW)$ since the start of the migration. We define the *violation fraction* VR_i as the fraction of sampling periods in which QoS contract violations occur: $VR_i = |\{k:L_i(k) > LC_i, k = 1,...,M\}| / M$.

1.3. Summary

The main contribution of the work we report here is a novel, control-theoretic approach to achieving these requirements. Our tool, *Aqueduct*, adaptively tries to consume as much as possible of the available system resources left unused by client applications while statistically avoiding QoS violations. It does so by dynamically adjusting the speed of data migration to maintain the desired QoS goals while maximizing the achieved data migration rate, using periodic measurements of the storage system's performance as perceived by the client applications. It guarantees that the average I/O latency throughout the execution of a migration will be bounded by a pre-specified QoS contract. If desired, it could be extended straightforwardly to provide a bound on the number of sampling periods during which the QoS contract was violated – but we found that it did so reasonably effectively without explicitly including this requirement, and suspected that doing so would reduce the data migration rate achieved – possibly more than was beneficial.

The focus in this paper is on providing latency guarantees because (1) our early work showed that bounds on latency are considerably harder to enforce than bounds on throughput – so a technique that could bound latency would have little difficulty with throughput; and (2) the primary beneficiaries of QoS guarantees are customer-facing applications, for which latency is a primary criterion.

To test Aqueduct, we ran a series of experiments on a state-of-the-art disk array [15] connected to a high-end host by a FibreChannel storage area network. Aqueduct successfully decreased the average I/O latencies of our client application workloads by as much as 76% compared with non-adaptive migration methods, successfully enforced the QoS contracts, and migrated data at close to the maximum speed allowed by the QoS guarantees. Although the current Aqueduct prototype does not explicitly guarantee a bound on the violation fraction VR_i, we nonetheless observed values of less than 17% in all our experiments.

The remainder of the paper contains a description of the Aqueduct system and our evaluation of it. We describe related work in Section 2, and the Aqueduct system architecture in Section 3. We then present the results of our experimental evaluation in Sections 4, 5 and 6, first describing our experimental infrastructure, and then the results of testing Aqueduct with two work-

loads: one purely synthetic, and the other a real, traced e-mail application. We conclude with some observations on our findings, and some suggestions for future work, in Section 7.

2. Related work

An old approach to performing backups and data relocations is to do them at night, while the system is idle. As discussed, this does not help with many current applications such as e-business that require continuous operation and adaptation to quickly changing system/workload conditions. The approach of bringing the whole (or parts of the) system offline is often impractical due to the substantial business costs it incurs.

Perhaps surprisingly, true online migration and backup are still in their infancy. But existing logical volume managers (e.g., the HP-UX logical volume manager, LVM [22], and the Veritas Volume Manager, VxVM [27]) have long been able to provide continuing access to data while it is being migrated. This is achieved by creating a mirror of the data to be moved, with the new replica in the place where the data is to end up. The mirror is then *silvered* – the replicas made consistent by bringing the new copy up to date – after which the original copy can be disconnected and discarded. Aqueduct uses this trick, too. However, we are not aware of any existing solution that bounds the impact of migration on client applications while this is occurring in terms that relate to their performance goals. Although VxVM provides a parameter, vol_default_iodelay, that is used to throttle I/O operations for silvering, it is applied regardless of the state of the client application. High-end disk arrays (e.g., the HP Surestore Disk Array XP512) provide restricted support for online data migration [14]: the source and destination devices must be identical Logical Units (LUs) within the same array, and only global, device-level QoS guarantees such as bounds on disk utilization are supported. Some commercial video servers [16] can re-stripe data online when disks fail or are added, and provide guarantees for the specific case of highly-sequential, predictable multimedia workloads. Aqueduct does not make any assumptions about the nature of the foreground workloads, nor about the devices that comprise the storage subsystem; it provides device-independent, application-level QoS guarantees.

Existing storage management products such as the HP OpenView-Performance Manager [13] can detect the presence of performance hot spots in the storage system when things are going wrong, and notify system administrators about them – but it is still up to humans to decide how to best solve the problem. In particular, there is no automatic throttling system that might address the root cause once it has been identified.

Although Aqueduct eagerly uses excess system resources in order to minimize the length of the migration, it is in principle possible to achieve zero impact on the foreground load by applying idleness-detection techniques [9] to migrate data only when the foreground load has temporarily stopped. Douceur and Bolosky [7] developed a feedback-based mechanism called MS Manners that improves the performance of important tasks by regulating the progress of low-importance tasks. MS Manners cannot provide guarantees to important tasks because it only takes as input feedback on the performance of the low-importance tasks. In contrast, Aqueduct provides performance guarantees to applications (i.e., the "important tasks") by directly monitoring and controlling their performance.

There has been substantial work on fair scheduling techniques since their inception [23]. In principle, it would be possible to schedule migration and foreground I/Os at the volume manager level without relying on an external feedback loop. However, real-world workloads are complicated and have multiple, nontrivial properties such as sequentiality, temporal locality, self-similarity, and burstiness. How to assign relative priorities to migration and foreground I/Os under these conditions is an open problem. For example, a simple 1-out-of-n scheme may work if the foreground load consists of random I/Os, but may cause a much higher than expected interference if foreground I/Os were highly sequential. Furthermore, any non-adaptive scheme is unlikely to succeed: application behaviors vary greatly over time, and failures and capacity additions occur very frequently in real systems. Fair scheduling based on dynamic priorities has worked reasonably well for CPU cycles; but priority computations remain an *ad hoc* craft, and the mechanical properties of disks plus the presence of large caches result in strong nonlinear behaviors that invalidate all but the most sophisticated latency predictions.

Recently, control theory has been explored in several computer system projects. Li and Nahrstedt [18] utilized control theory to develop a feedback control loop to guarantee the desired network packet rate in a distributed visual tracking system. Hollot et al. [11] applied control theory to analyze a congestion control algorithm on IP routers. While these works apply control theory on computing systems, they focus on managing the network bandwidth instead the performance of end servers.

Feedback control architectures have also been developed for web servers [6][19] and e-mail servers [25]. In the area of CPU scheduling, Steere et al. [26] developed a feedback based CPU scheduler that synchronize the progress of consumers and supplier processes of buffers. In [20], scheduling algorithms based on feedback control were developed to provide deadline miss ratio guarantees to real-time applications with unpredictable workloads. Although these approaches show clear promise, they do not guarantee I/O latencies to applications, nor do they address the storage subsystem, which is the focus of Aqueduct. The feedback-based web cache manager described in [21] achieves differentiated cache hit ratio by adaptively allocating storage spaces to user classes. However, they also did not address I/O latency or data migration in storage systems.

3. Aqueduct

The overall architecture of Aqueduct, and the way in which it interacts with its environment, are shown in Figure 1. It takes in a QoS contract and a migration plan, and interacts with the storage system to migrate data by using the existing HP-UX LVM's [22] primitives. As discussed above, Aqueduct relies upon the LVM silvering operation to achieve a move without having to disable client application accesses.

Ideally, Aqueduct would be integrated with the LVM, so that it could directly control the rate at which data were moved in order to achieve its QoS guarantees. Alternatively, if a dynamically-alterable parameter had been provided to control LVM's silvering rate (i.e., data movement speed), Aqueduct could have used that to effect its control. Unfortunately, neither was possible for our experiments, so we resorted to a (somewhat crude) approximation to these more tightly-coupled approaches. Fortunately, despite the overheads it imposed, it proved adequate to our goal of confirming the potential benefits of the control-feedback loop approach.

Aqueduct divides each store into small, fixed-size *substores* that are migrated one at a time, in steps called *submoves*. This allows relatively fine control over the migration speed, as substores are relatively small: we chose 32MB as a reasonable compromise between management overheads and control finesse. With LVM, we were forced to implement each substore as a complete logical volume in its own right; unfortunately, this had the undesirable property that store migrations were visible at the application level. (VxVM might have let us lift this restriction, but we did not have easy access to a running implementation.) The resulting large numbers of logical volumes incur considerable LVM-related overheads. Nonetheless, despite the overheads, this implementation allowed us to evaluate the key part of the Aqueduct architecture – the feedback control loop – which was the primary point of this exercise.

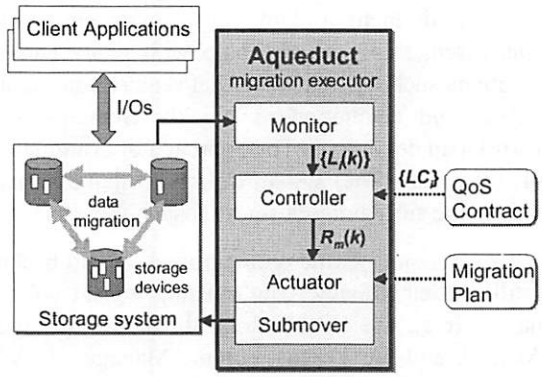


Figure 1 The internal structure of the Aqueduct migration executor, and its relationship to the external world.

3.1. Feedback control loop in Aqueduct

The Aqueduct *monitor* component is responsible for collecting the sampled latency of each store at the end of each sampling period, and feeding these results to the *controller*. We were able to extract this data directly from an output file periodically generated by our workload generation tool; but it could also have been obtained from other existing performance monitoring tools (e.g., the GlancePlus tool of HP OpenView [13]).

The *controller* compares the sampled latencies for the time window $((k\text{-}1)W, kW)$ with the QoS contract, and computes the submove rate $R_m(k)$ (the *control input*) to be used during the next sampling period $(kW, (k+1)W)$. Intuitively, Aqueduct should slow down data migration when some sampled store latencies are larger than their corresponding contracts, and speed up data migration when latencies are smaller than the corresponding contracts for all stores. The controller computes the submove rate based on the sampled store latencies so that the sampled store latencies stay close to their corresponding contracts. Aqueduct incorporates an *integral controller*, a well-studied law in control theory [8]; integral controllers are typically robust in the presence of a wide range of workload variations. It operates as follows:

1) For each store i $(0 \le i < N)$ in the system, compute its error

$$E_i(k) = P*LC_i - L_i(k),$$

where P ($0 < P < 1$) is a configurable parameter, and $P*LC_i$ is called the *reference* in control theory. More negative values of $E_i(k)$ represent larger latency violations.

2) Find the smallest (i.e., potentially most negative) error $E_{min}(k)$ among all stores:
$E_{min}(k) = \min\{E_i(k) | 0 \le i < N\}$;
thus taking account of the worst contract violation observed.

3) Compute the submove-rate according to the integral control function (K is another configurable parameter of the controller):
$R_m(k) = R_m(k\text{-}1) + K*E_{min}(k)$;

4) Notify the actuator of the new submove rate $R_m(k)$.

Because the control input $R_m(k)$ is computed from the $E_i(k)$ corresponding to the worst violation, it forces the system to satisfy its latency goals by arranging for E_{min} to converge to zero. Thanks to random workload variations, store I/O latencies will typically oscillate around the reference value, so instead of choosing the actual latency target LC_i as the reference, the controller uses a slightly smaller target: $P*LC_i$. The value of P is related to the burstiness of the workload: the more bursty a workload is, the smaller P should be, to give the controller enough leeway to avoid contract violations. On the other hand, overly small values of P will result in an overly conservative controller, and therefore slow down migration. In our experiments, we observed that a P between 0.8 and 0.9 was sufficient to achieve satisfactory violation fractions for significantly different workloads.

Parameter K needs to be tuned to achieve *stability* (i.e., to prevent the submove rate and sampled latencies from oscillating excessively) and *short settling time* (i.e., fast convergence of the output to the reference). This can be done using systematic, standard control theory techniques. An example is provided in Section 5.1. A similar tuning method was described in detail, and applied to a real-time CPU scheduler in [20]. Aqueduct could be extended in a fairly straightforward way to set (and adjust) K automatically, using an online estimation of the gain [5] in order to handle different categories of workloads without the need for precomputed parameter values.

The last module in Figure 1 is the *actuator*. It executes a migration plan at the submove rate computed by the controller. During the sampling period (kW, $(k+1)W$), the actuator enforces the submove rate $R_m(k)$ by sleeping for ($W/R_m(k)$ - T_j) time units between the end of submove j and the start of the next, where T_j is the time it took to complete submove j.

4. Experiment overview

We evaluated the performance of Aqueduct in our storage area network, using both a synthetic workload and an I/O trace from a production e-mail server.

The hardware used for our tests includes an HP FC-60 disk array [15] with 512 MB of cache, two redundant controllers, and six disk enclosures with 5 disks each, for a total unprotected capacity of 1.05TB. All LUs in the array were 6-disk RAID-5s with 16-KB stripe units. The FC-60 array was connected to a Brocade Silkworm 2800 switch via two 1Gb/s Fibre Channel links. Both Aqueduct and the load generators ran on the same HP 9000-N4000 server, which has eight 440 MHz PA-RISC 8500 processors and 16GB of RAM. The host ran the HP-UX 11.0 operating system. We used our own workload-generation tool, Buttress, which is capable of generating synthetic workloads and replaying an existing I/O trace with very high fidelity.

We compared Aqueduct against two baselines:

- **Whole-store**, moves a whole store in each step as fast as possible, with no delays between store-moves; stores are not divided into smaller sub-stores. This is designed to reflect what a system administrator would do when migrating data by hand or by running simple scripts.
- **Sub-store** is similar to Whole-store, but divides each store into substores and performs each move as a sequence of submoves. It is a fairer baseline for comparison with Aqueduct because it uses the same number of logical volumes (substores) and hence incurs similar amounts of LVM overhead.

In these experiments, all stores were given the same latency contract, LC = 10 ms, and we always used a sampling period (W), of 60 seconds, and a substore size of 32MB. The following table lists the configurable parameters we used for the two workloads:

	Synthetic	*OpenMail*
K	*1.09*	*0.36*
P	*0.90*	*0.80*

Since the OpenMail workload is more sensitive to changes in the submove rate than the synthetic workload, we tuned K to be smaller for the OpenMail workload based on control theory (described in Section 5.1). We found that the first value we tried for P (0.9) was adequate for the synthetic workload, but not for OpenMail—the second trial for OpenMail resulted in the final, slightly smaller P = 0.8. This was not unexpected, as OpenMail is more bursty than our deliberately well-behaved synthetic workload.

We define the *victim latency* $VL(k)$ as the highest sampled latency of all stores in the k^{th} sampling period, i.e., $VL(k) = \max\{L_i(k): 0 \le i < N\}$. In this special case in which all stores share a same latency contract LC, all stores satisfy their contracts if and only if the victim latency stays lower than the contract. Similarly, the *average victim latency* AVL is the average of the values of $VL(k)$ over all M sampling periods during the migration. AVL reflects the "correctness" of the migration speed. Ideally, AVL should be close to the latency contract. If $AVL > LC$, the migration runs too fast and causes excessive contract violations. On the other hand, if $AVL < LC$, the migration could have run faster without violating the latency contract.

5. Synthetic workload experiments

Figure 2 illustrates the initial state and the migrations that were effected in this test. The synthetic workload is composed of multiple streams with fixed, identical parameters, and tests Aqueduct in the presence of deliberately steady workloads. We configured two LVM volume groups, aq0 and aq1. All stores are 640 MB in size, and are therefore divided into 20 substores each. Group aq0 contains six stores. In the initial assignment, three of them (the *migrate-stores* M0, M1, M2) are mapped onto logical unit LU1 of the disk array; the remaining three stores of aq0 (the *fixed-stores* F0, F1, F2) are in another logical unit LU0.

These experiments emulate the following use scenario: assume that we find that LU1 is likely to fail (e.g., by using system monitoring tools such as [17]), so we want to move the migrate-stores in LU1 to a new logical unit LU3. Hence, the migrate- and the fixed-stores belong to the same LVM volume group. We hypothesized that stores contained within the same volume group where data is being migrated would suffer some performance penalty from LVM overhead, even if they were not being migrated. To test that assumption, we created group aq1 on LU2, whose stores (the *alone-stores* A0, A1, A2) should not be affected because they are totally separate from aq0.

To generate the workload from the client applications, we simulated the file system workload described in [2] by issuing two synthetic streams on each store. Each stream has a Poisson arrival process, 16KB request size, with a run count of 3 (this is the average number of consecutive I/O requests performed on consecutive addresses), 64% of the requests are reads, and the request rate is 32/second.

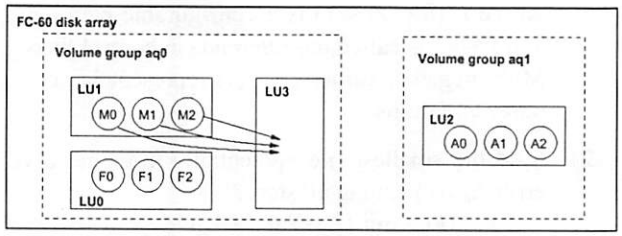


Figure 2 Placement of stores on disk array logical units for the synthetic workload, and the migration that was performed.

5.1. Tuning *K*

We demonstrate here how the parameter K was selected for the synthetic workload using the Root-Locus design technique [8]. The *controlled system* includes the storage system, the monitor, and the actuator. The output is the victim latency $VL(k+1)$, i.e., the sampled latency of the store with the smallest error $E_{min}(k+1)$ in the sampling period $(kW, (k+1)W)$. The input to the controlled system is the submove rate $R_m(k)$ in the sampling period $(kW, (k+1)W)$. ($R_m(k)$ instead of $R_m(k+1)$ is used to denote the submove rate in $(kW, (k+1)W)$ because the controller outputs $R_m(k)$ at time kW instead of $(k+1)W$). In our design, we approximate the controlled system with the following linear model:

$$VL(k+1) - VL(k) = G(R_m(k) - R_m(k-1)) \qquad (1)$$

The *process gain*, G, is the derivative of the output $VL(k+1)$ with respect to the input $R_m(k)$. G represents the sensitivity of the victim latency with regard to the change in submove rate. We approximate G by running a set of system profiling experiments. In each run, migration is performed at a fixed submove rate, and different submove rates are used in different runs. The average victim latencies observed throughout migration for different submove rates are plotted in Figure 3. Using linear regression, we estimate that $G = 1.12$ with an R^2 of 99% for the synthetic workload.

We now transform the controlled system model into the z-domain, which is amenable to control analysis. The controlled system model in Equation 1 is equivalent to the following transfer function from $R_m(z)$ to $VL(z)$ in z-domain: $H(z) = G*z^{-1}$. The integral controller is transformed to the following transfer function from the minimum error $E_{min}(z)$ to the submove rate, $R_m(z)$, in the z-domain: $C(z) = Kz/(z-1)$.

It follows that the whole feedback control system composed of the controlled system and the integral controller is modeled as the following transfer function from the reference to the victim latency:

$$H_C(z) = \frac{C(z)H(z)}{1 + C(z)H(z)} \quad (2)$$

Assume that all the stores share a common contract $P*LC$, the z-transform of the victim latency is:

$$VL(z) = H_C(z) * P * LC * \frac{z}{z-1} \quad (3)$$

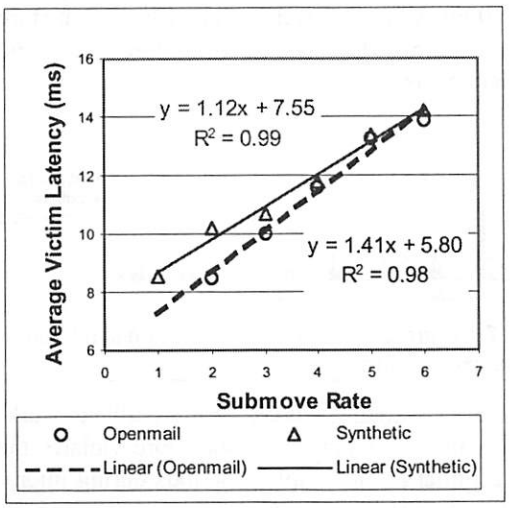


Figure 3 Victim latency as a function of submove rate for the synthetic workload and openmail workload.

Given the dynamic model of the closed loop system, we tune the control parameter K analytically using linear control theory [8], which states that the performance of a system depends on the poles of its closed loop transfer function. Since the closed loop transfer function (Equation 2) of Aqueduct has a single pole $p = 1-KG$, we can set p to the desired value by choosing the right value of K. The sufficient and necessary condition for Aqueduct to guarantee stability is: $|p| < 1 \Leftrightarrow 0 < K < 2/G$. The settling time represents the time it takes to converge the victim latency to the reference. A smaller settling time leads to a faster response to workload variations. The settling time is determined by the damping ratio of the closed loop system. A larger G (e.g., in a workload whose latency is more sensitive to the submove rate) needs a smaller K to get the same pole and achieve the same level of stability and settling time. Using the root-locus method, we set $p = -0.22$ by choosing $K = (1-p)/G = 1.09$ to guarantee stability and a short settling time. The OpenMail has a larger gain than the synthetic workload, so it benefits from a lower value of K.

5.2. Experimental results

We now explore the results of applying Aqueduct to migrating data that is being accessed by the synthetic workload.

The sampled latencies of store M0 in typical runs of Aqueduct and the baselines are illustrated in 0. (We pick M0 because it is the store most affected by migration, as shown in Figure 6.) Whole-store causes long latencies throughout migration; latencies are especially severe near the end of the migration when they jump to 25.17 ms. This is because near the end of the migration, more application I/Os target at the new logical unit, LU3, and contend more severely with Whole-store which *writes* into LU3 in parallel. Interestingly, in the beginning of migration, more application I/Os target at the replaced logical unit LU1 and contend with Whole-store which *reads* data from LU1, but the impact of migration on latencies is less severe. This is because writes are more expensive than reads on RAID5 (especially if they are small, as done by LVM when silvering), and therefore migration consumes more resources on LU3 than on LU1. Although Whole-store completes data migration within the shortest time, it violates the latency contract (10 ms) throughout the data migration period.

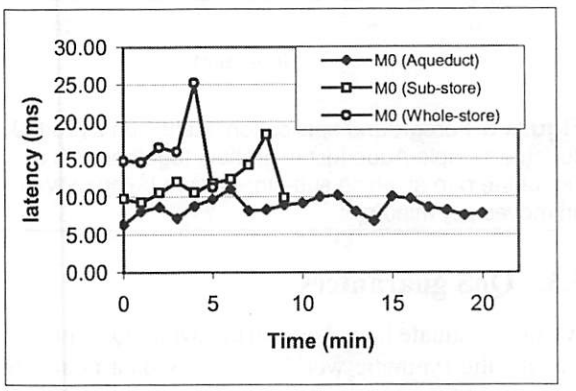


Figure 4 Data from three typical runs in the synthetic workload experiments, showing foreground application I/O latency on store M0 during the execution of the Aqueduct, Sub-store, and Whole-store migration algorithms.

Sub-store migrates data more slowly, and with smaller impact on client applications, than Whole-store. However, the latency contracts are still violated in most sampling periods. Since neither Sub-store nor Whole-store sleeps between subsequent (sub)moves, we attribute the difference between their migration times and interference on client applications to the overhead of managing large number of logical volumes in Sub-store—which is slowed down by this effect.

In comparison, in the case of Aqueduct, the latency of M0 stays below the latency contract in most of the sampling periods. This result demonstrates that Aqueduct effectively reduces migration's impact on client applications. Note that the latency of M0 stays close to

the contract latency. This indicates that, although Aqueduct has a longer migration time than the baselines, it achieves a submove rate that is close to the maximum allowed by the QoS contract.

To demonstrate the quality of control by Aqueduct, we plot the traces of sampled latency on M0 and submove rate (the control input) during the same sample run. We can see that Aqueduct effectively keeps latency close to the reference (9 ms) by dynamically adapting the submove rate—peaks and valleys are strongly correlated in the two curves. Furthermore, Aqueduct achieves satisfactory stability because it does not cause excessive oscillation in submove rate or latency throughout the run.

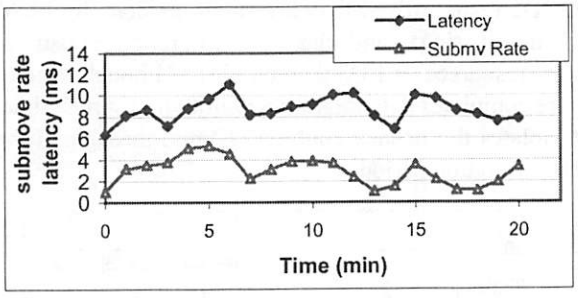


Figure 5 Foreground application latency on store M0 during a sample Aqueduct migration, together with a plot of the rate at which sub-stores were being moved (in moves per minute).

5.3. QoS guarantees

We now evaluate how Aqueduct provides QoS guarantees for the synthetic workload. Every data point presented in this section and Section 5.4 is the mean of 5 repeated runs. We also report the 90% confidence intervals for every data point.

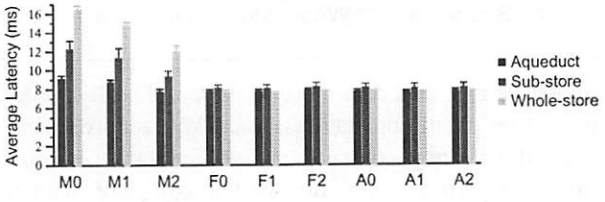


Figure 6 Average application I/O latencies in the synthetic workload experiments.

The average latencies for Aqueduct and the baselines are illustrated in Figure 6. The latencies of the alone-stores and the fixed-stores are similar, and therefore the impacts of LVM overhead on fixed-stores are negligible. Migration has negligible impacts on the average latencies of the fixed-stores or the alone-stores with all migration methods. However, different migration methods perform differently on the migrated stores. In particular, Whole-store achieves an average latency on M0 of 16.4 (±0.5) ms,, which is 80% higher than Aqueduct's 9.1 (±0.4) ms. Similarly, Sub-store achieves an average latency of 12.2 (±0.9) ms, or 34% higher than Aqueduct. More importantly, Aqueduct's average latencies of all stores are lower than the latency contract of 10 ms, while the average latencies of Sub-store and Whole-store are higher than the contract in two and three migrate-stores, respectively.

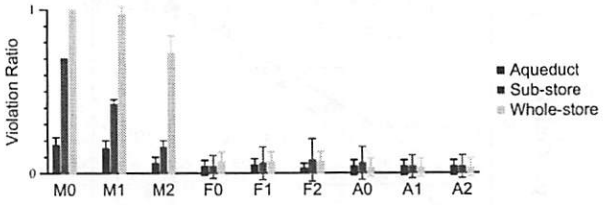


Figure 7 Contract violation fractions in the synthetic workload experiments.

The contract violation fractions for the synthetic workload are shown in Figure 7. Whole-store violates the latency contract in all sampling periods during migration. While Sub-store achieves a lower contract violation fraction due to LVM overheads in data migration, it still causes a much higher violation fraction than Aqueduct. In particular, Sub-store violates the latency contract in 70% (±0%) of all sampling periods during migration, while Aqueduct only violates 17% (±5%) of all sampling periods. The contract violation fraction is important because a lower value means that client applications suffer violations less frequently and hence the storage service has more acceptable performance.

5.4. Migration efficiency

As expected, Aqueduct provides QoS guarantee to applications at the expense of slowing down data migration. Figure 8a shows that it takes Aqueduct 1219 (±43) sec on average to complete the migration plan, while Sub-store only needs 556 (±3) sec. Sub-store migrates data more slowly than Whole-store due to the LVM overhead.

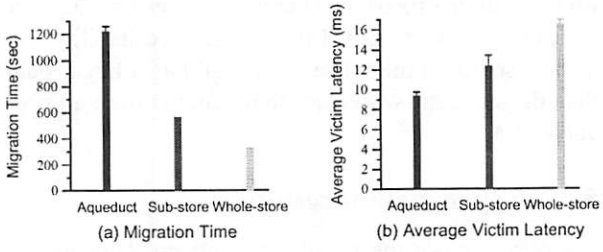


Figure 8 Migration times in synthetic workload experiments.

In order to determine whether Aqueduct achieves the maximum speed allowed by the QoS contracts, we look at average victim latencies. The rationale is that, to guarantee that no stores violate the latency contract, the victim latency must be the same or lower than the latency contract, i.e., the latency contract must be an upper bound on the victim latency. Figure 8b shows that Aqueduct achieves an average victim latency of 9.30 (±0.46) ms, which is only 7% lower than the latency contract. Given that the average submove rate is below 3 submoves/min., even an increase of 1 submove/min. in the control input would result in service contract violations. This result shows that Aqueduct's bound is tight: Aqueduct is not overly conservative, and it achieves a migration speed close to the maximum that is possible given the constraint of providing latency guarantees. In addition, note that the average victim latency is close to the actual reference to the controller (P^*LC = 9 ms), which shows that the controller correctly enforces the reference.

In summary, the synthetic-workload experiments demonstrate that Aqueduct can effectively provide latency guarantees to applications having steady, regular access patterns, while performing online data migration efficiently. Aqueduct guarantees the average latencies of all stores to be lower than the latency contract, and achieves a contract violation fraction of no more than 17%. For the same migration plan, Whole-store and Sub-store cause average latencies higher than the latency contract in migrated-stores and contract violations as high as 100% and 70%, respectively. In term of migration efficiency, Aqueduct achieves a migration speed close to the maximum allowed by the latency contract.

6. OpenMail experiments

The OpenMail workload was originally gathered by tracing an e-mail server running HP OpenMail [12]. The original workload trace was collected on an HP 9000 K580 server system with an I/O subsystem comprised of four EMC Symmetrix 3700 disk arrays. The server was sized to support a maximum of about 4500 users, although only about 1400 users were actively accessing their email during the trace collection period, which corresponded to the server's busiest hour of the day. The majority of accesses in the trace are to the 640 GB message store, which is striped uniformly across all of the arrays.

In order to create a trace comparable to our synthetically generated workloads, we replayed the portion of the original trace corresponding to a single representative array on our FC-60 array. Since the LVM on HP-UX 11.0 has a limitation that each volume group can contain at most 255 logical volumes, and each logical volume corresponds to one substore (32 MB each) in our current Aqueduct prototype, we shrank the sizes of the corresponding stores proportionally to a total size of 3.8 GB to fit them into one volume group. (This size limitation can be fixed by a future Aqueduct implementation with modifications on the LVM.)

This workload has significantly more complex behaviors than our synthetic one. The OpenMail system being traced kept a small amount of metadata (an index table) at the beginning of the message store's address space, and filled up the remainder with e-mail messages. For each email retrieval request from a user, or on each incoming email, the server accesses the initial index table and then jumps to actually access the message, to a random location uniformly distributed across the upper portion of the store. Consequently, the small amount of metadata becomes a hotspot that gets accessed much more frequently than the other data.

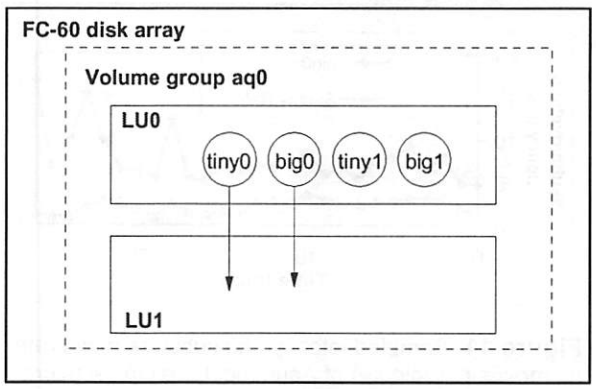


Figure 9 Store migration plan for the OpenMail workload.

We create one volume group, aq0, which includes 4 stores called tiny0, tiny1, big0, and big1, respectively. tiny0 and tiny1 are 96 MB each, and big0 and big1 are 1854 MB each. In the initial assignment, all the stores are located on a single logical unit LU0.

The OpenMail experiments emulate a LU-addition scenario. We model the case of wanting to increase the server capacity by adding a new Logical Unit, LU1, to the array. To make use of the new LU, we migrate two stores, tiny0 and big0, from LU0 to LU1.

Similarly to the synthetic workload, we approximate the process gain, G, for the openmail workload with a set of system profiling experiments. In each run, migration is performed at a fixed submove rate, and different submove rates are used in different runs. The average victim latencies observed throughout migration

for different submove rates are plotted in Figure 3. Using linear regression, we estimate that $G = 1.41$ with an R^2 of 98% for the openmail workload. Compared with the synthetoc workload, the process gain of the openmail workload larger. This result means that openmail is more senstive to the impacts of migration and therefor a smaller K is needed. In our experiments we set $K = 0.36$ (corresponding to a pole $p = 0.49$) to guarantee stability and a short settling time for the openmail workload.

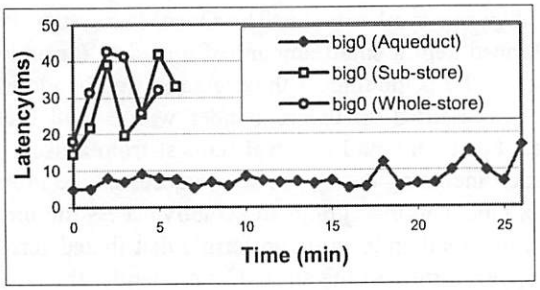


Figure 10 Sampled latency for the OpenMail workload during migration.

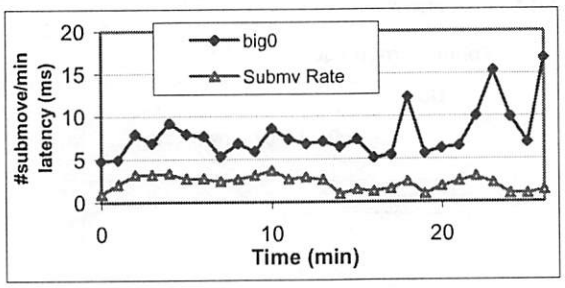


Figure 11 Sampled latency and substore move rate (in moves per minute) of Aqueduct for a typical migration of the OpenMail workload.

The sampled latencies of store big0 in typical runs of Aqueduct and the baselines are illustrated in Figure 10. Both Whole-store and Sub-Store cause extremely long latencies on big0 and violate the latency contract throughout migration. In comparison, with Aqueduct, big0's latencies stay below the latency contract (10 ms) in most sampling periods. Figure 11 shows the traces of sampled latency on big0 and submove rate during the same sample run. Aqueduct effectively keeps latency close to the reference (8 ms) by dynamically adapting the submove rate without causing excessive oscillation.

In the following subsections, we present the detailed evaluation results of Aqueduct in the OpenMail experiments. Every data point presented in this section is the mean of five repeated runs. The 90% confidence intervals are also plotted.

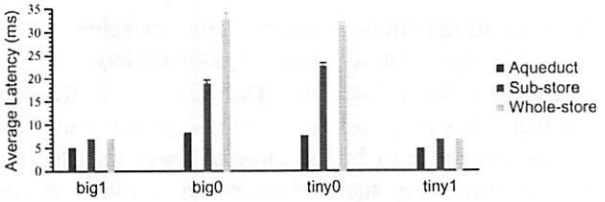


Figure 12 Average application I/O latencies during migrations for the OpenMail workload.

6.1. QoS guarantees

The average latencies in the OpenMail experiments are shown in Figure 12. The OpenMail application is much more sensitive to data migration overheads than the synthetic workload. For example, Sub-store increases the average latencies of accesses to the migrated-stores, big0 and tiny0, to 18.74 (±0.92) ms and 22.46 (±0.81) ms – which are 87% and 125% higher than the latency contract (10 ms), respectively. In comparison, Aqueduct achieves an average latency no higher than 7.70 (±0.36) ms, or 23% lower than the contract in all stores. This result demonstrates the efficacy of Aqueduct in applications that are very vulnerable to online data migrations.

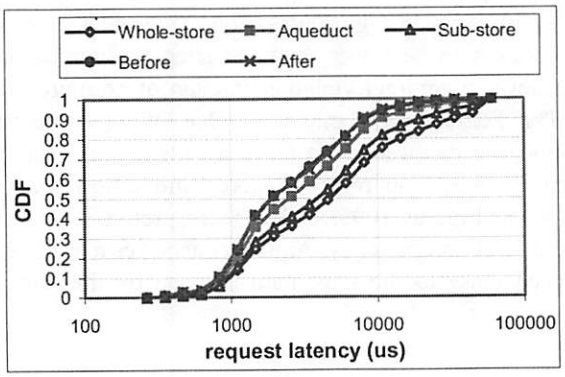


Figure 13 Cumulative distribution of I/O times during data migration by the three different schemes for the OpenMail workload, across the entire workload. Aqueduct has the smallest impact on the I/O request latencies. The "before" and "after" values on this plot are for the sub-store case, but the differences with the other alternatives are almost too small to show. Note the log scale on the x-axis.

A study of the distribution of I/O request latencies during a migration (Figure 13) shows that the effect of Aqueduct is to reduce the number of requests that suffer significantly longer I/O times: application I/Os queued behind a data migration operation result in large delays.

The contract violation fractions for the different migration algorithms are shown in Figure 14. Aqueduct sig-

nificantly reduces the contract violation fractions of the migrated stores, big0 and tiny0. For example, the contract violation fraction of tiny0 is reduced from 98% with Sub-store to only 7% with Aqueduct. Thus, Sub-store causes applications to suffer contract violations in almost every sampling period during data migration, while contract violations rarely occur with Aqueduct.

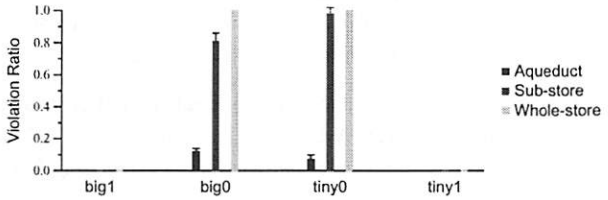


Figure 14 Contract violation fractions in OpenMail experiments.

6.2. Migration efficiency

As shown in Figure 15a, Aqueduct increases the migration time more significantly in the case of OpenMail than in the case of the synthetic workload. Because OpenMail is affected more severely by migration, Aqueduct is forced to perform migration more slowly.

The average victim latency (see Figure 15b) of Aqueduct is 8.46 (±0.31) ms, or 15% lower than the latency contract. Again, the migration speed is close to the maximum speed allowed by the latency contract. We also note that the average victim latency is within 6% of the reference (8 ms), which shows that the Aqueduct controller is able to successfully track the control reference even in the presence of bursty workloads such as OpenMail.

In summary, the OpenMail experiments demonstrate that Aqueduct provides latency guarantee to real-world applications that are especially sensitive to migration. In particular, Aqueduct meets its QoS guarantees, and achieves an average victim latency that is only 15% below the latency contract. As in the synthetic experiments, Aqueduct performs migration at a speed close to the maximum allowed by the latency contract.

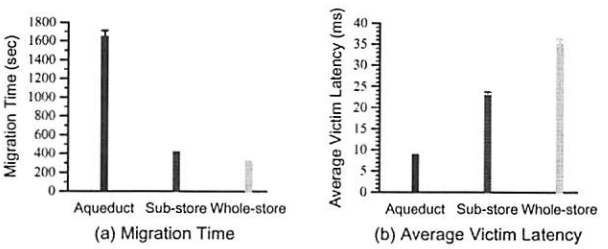


Figure 15 Migration time in OpenMail experiments.

7. Conclusions and future work

We have developed Aqueduct, an online data migration architecture that provides QoS guarantees to client applications. Aqueduct features a feedback control loop that dynamically adapts migration speed to maintain performance guarantees in the presence of workload and system variations.

We evaluated a prototype on a real storage system, using a high-end host and disk arrays similar to the ones used in large enterprise installations. Our experiments show that Aqueduct successfully provides QoS guarantees in term of bounded average latencies, while causing only a small percentage of contract violations. Aqueduct reduces the average I/O latency experienced by client applications by as much as 76% with respect to the traditional method: while accesses to a store in an e-mail server have an average I/O latency of 32.6 ms while a non-adaptive migration is in progress, accesses to the same store have an average latency of only 7.7 ms with Aqueduct. Aqueduct also reduces the violation fraction from 100% to only 12%. Furthermore, Aqueduct performs data migration very close to the maximum speed allowed by the latency contract, as evidenced by the small slack of only 15% between the average victim latency and the latency contract.

Potential future work items include a more general implementation that interacts with performance monitoring tools, developing a low overhead mechanism for finer-grain control of the migration speed, making the controller self-tuning to handle different categories of workloads, and implementing a new control loop that can simultaneously bound latencies and violation fractions.

Acknowledgements: we thank Eric Anderson, Michael Hobbs, Kimberly Keeton, and Mustafa Uysal for clarifying many infrastructure-related questions. We also want to thank Jack Stankovic for helpful comments on an earlier version of this paper. The FAST referees and our shepherd, Roger Haskin, have helped us to improve this paper with their useful feedback.

References

[1] 3Com Corporation, "Gigabit Ethernet Comes of Age," Technology white paper, June 1996.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. "Minerva: an automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, vol. 19, no. 4, November 2001.

[3] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. "Experimental Study of Data Migration Algorithms," *5th Workshop on Experimental Algorithms*, August 2001.

[4] ANSI, "Fibre Channel Arbitrated Loop," Standard X3.272-1996, April 1996.

[5] T. F. Abdelzaher, "An Automated Profiling Subsystem for QoS-Aware Services," *IEEE Real-Time Technology and Applications Symposium*, pp. 208-217, June 2000.

[6] T. F. Abdelzaher and N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," *International Workshop on Quality of Service*, pp. 216-225, 1999.

[7] J. R. Douceur and W. J. Bolosky. "Progress-based regulation of low-importance processes," *17th ACM Symposium on Operating Systems Principles*, pp. 247-260. Dec 1999.

[8] G. F. Franklin, J. D. Powell and M. L. Workman, *Digital Control of Dynamic Systems (3rd Ed.)*, Addison-Wesley, 1998.

[9] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and John Wilkes. "Idleness is not Sloth," *Winter'95 USENIX Conference*, pp 201-212, Jan. 1995.

[10] S. Gribble, G. Manku, E. Roselli and E. Brewer, "Self-similarity in File Systems," *SIGMETRICS'98*, pp. 141-150, April 1998.

[11] C. V. Hollot, V. Misra, D. Towsley, and W. Gong, "A Control Theoretic Analysis of RED," *IEEE INFOCOM*, pp. 1510-1519, April 2001.

[12] Hewlett-Packard Company, *Openmail*, http://www.OpenMail.com/cyc/om/00/index.html.

[13] Hewlett-Packard Company, *HP OpenView Homepage*, http://www.openview.hp.com/.

[14] Hewlett-Packard Company, *HP SureStore E Auto LUN XP User's Guide*, May 2000.

[15] Hewlett-Packard Company, *HP SureStore E Disk Array FC60 – Advanced User's Guide*, Dec 2000.

[16] International Business Machines Corp., Content Manager VideoCharger Administrator's Guide, January 2001.

[17] International Business Machines Corp., *Drive Fitness Test*, August 1999.

[18] B. Li and K. Nahrstedt, "A Control-based Middleware Framework for Quality of Service Adaptations," *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 17(9), pp. 1632-1650, Sept. 1999.

[19] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers," *IEEE Real-Time Technology and Applications Symposium*, pp. 51-62, June 2001.

[20] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms," *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 22(3), March 2002.

[21] Y. Lu, A. Saxena, and T. F. Abdelzaher, "Differentiated Caching Services; A Control-Theoretical Approach," *International Conference on Distributed Computing Systems*, pp. 615-622, April 2001.

[22] T. Madell, *Disk and File Management Tasks in HP-UX*, Prentice-Hall, 1997.

[23] J. Nagle. "On packet switches with infinite storage," *IEEE Trans. on Communications*, vol. 35, no. 4, pp. 435-38, April 1987.

[24] G. Papadopoulos, "Moore's Law Ain't Good Enough," keynote speech at *Hot Chips X*, August 1998.

[25] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, J. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[26] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, J. Walpole, "A Feedback-driven Proportion Allocator for Real-Rate Scheduling," *Symposium on Operating Systems Design and Implementation,* pp. 145-158, Feb 1999.

[27] Veritas Software Corp., *Veritas Volume Manager*, http://www.veritas.com/us/products/volumemanager.

GPFS: A Shared-Disk File System for Large Computing Clusters

Frank Schmuck and Roger Haskin
IBM Almaden Research Center
San Jose, CA

Abstract

GPFS is IBM's parallel, shared-disk file system for cluster computers, available on the RS/6000 SP parallel supercomputer and on Linux clusters. GPFS is used on many of the largest supercomputers in the world. GPFS was built on many of the ideas that were developed in the academic community over the last several years, particularly distributed locking and recovery technology. To date it has been a matter of conjecture how well these ideas scale. We have had the opportunity to test those limits in the context of a product that runs on the largest systems in existence. While in many cases existing ideas scaled well, new approaches were necessary in many key areas. This paper describes GPFS, and discusses how distributed locking and recovery techniques were extended to scale to large clusters.

1 Introduction

Since the beginning of computing, there have always been problems too big for the largest machines of the day. This situation persists even with today's powerful CPUs and shared-memory multiprocessors. Advances in communication technology have allowed numbers of machines to be aggregated into computing *clusters* of effectively unbounded processing power and storage capacity that can be used to solve much larger problems than could a single machine. Because clusters are composed of independent and effectively redundant computers, they have a potential for fault-tolerance. This makes them suitable for other classes of problems in which reliability is paramount. As a result, there has been great interest in clustering technology in the past several years.

One fundamental drawback of clusters is that programs must be partitioned to run on multiple machines, and it is difficult for these partitioned programs to cooperate or share resources. Perhaps the most important such resource is the file system. In the absence of a cluster file system, individual components of a partitioned program must share cluster storage in an ad-hoc manner. This typically complicates programming, limits performance, and compromises reliability.

GPFS is a parallel file system for cluster computers that provides, as closely as possible, the behavior of a general-purpose POSIX file system running on a single machine. GPFS evolved from the Tiger Shark multimedia file system [1]. GPFS scales to the largest clusters that have been built, and is used on six of the ten most powerful supercomputers in the world, including the largest, ASCI White at Lawrence Livermore National Laboratory. GPFS successfully satisfies the needs for throughput, storage capacity, and reliability of the largest and most demanding problems.

Traditional supercomputing applications, when run on a cluster, require parallel access from multiple nodes *within* a file shared across the cluster. Other applications, including scalable file and Web servers and large digital libraries, are characterized by *interfile* parallel access. In the latter class of applications, data in individual files is not necessarily accessed in parallel, but since the files reside in common directories and allocate space on the same disks, file system data structures (metadata) are still accessed in parallel. GPFS supports fully parallel access both to file data and metadata. In truly large systems, even administrative actions such as adding or removing disks from a file system or rebalancing files across disks, involve a great amount of work. GPFS performs its administrative functions in parallel as well.

GPFS achieves its extreme scalability through its *shared-disk* architecture (Figure 1) [2]. A GPFS system consists of the cluster nodes, on which the GPFS file system and the applications that use it run, connected to the disks or disk subsystems over a switching fabric. All nodes in the cluster have equal access to all disks. Files are striped across all disks in the file system – several thousand disks in the largest GPFS installations. In addition to balancing load on the disks, striping achieves the full throughput of which the disk subsystem is capable.

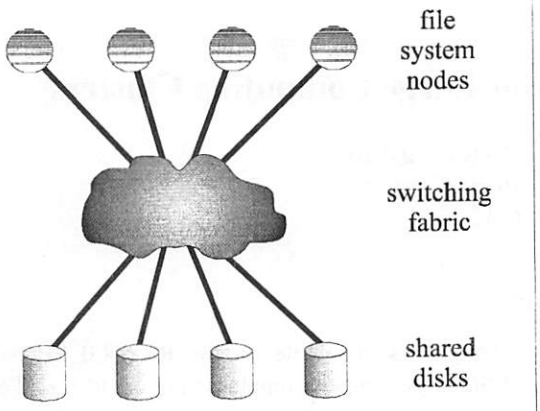


Figure 1: Shared Disk Environment

The switching fabric that connects file system nodes to disks may consist of a storage area network (SAN), e.g. fibre channel or iSCSI. Alternatively, individual disks may be attached to some number of I/O server nodes that allow access from file system nodes through a software layer running over a general-purpose communication network, such as IBM's Virtual Shared Disk (VSD) running over the SP switch. Regardless of how shared disks are implemented, GPFS only assumes a conventional block I/O interface with no particular intelligence at the disks.

Parallel read-write disk accesses from multiple nodes in the cluster must be properly synchronized, or both user data and file system metadata will become corrupted. GPFS uses distributed locking to synchronize access to shared disks. GPFS distributed locking protocols ensure file system consistency is maintained regardless of the number of nodes simultaneously reading from and writing to the file system, while at the same time allowing the parallelism necessary to achieve maximum throughput.

This paper describes the overall architecture of GPFS, details some of the features that contribute to its performance and scalability, describes its approach to achieving parallelism and data consistency in a cluster environment, describes its design for fault-tolerance, and presents data on its performance.

2 General Large File System Issues

The GPFS disk data structures support file systems with up to 4096 disks of up to 1TB in size each, for a total of 4 petabytes per file system. The largest single GPFS file system in production use to date is 75TB (ASCI White [3, 4, 5]). GPFS supports 64-bit file size interfaces, allowing a maximum file size of 2^{63}-1 bytes. While the desire to support large file systems is not unique to clusters, the data structures and algorithms that allow GPFS to do this are worth describing.

2.1 Data Striping and Allocation, Prefetch and Write-behind

Achieving high throughput to a single, large file requires striping the data across multiple disks and multiple disk controllers. Rather than relying on a separate logical volume manager (LVM) layer, GPFS implements striping in the file system. Managing its own striping affords GPFS the control it needs to achieve fault tolerance and to balance load across adapters, storage controllers, and disks. Although some LVMs provide similar function, they may not have adequate knowledge of topology to properly balance load. Furthermore, many LVMs expose logical volumes as logical unit numbers (LUNs), which impose size limits due to addressability limitations, e.g., 32-bit logical block addresses.

Large files in GPFS are divided into equal sized blocks, and consecutive blocks are placed on different disks in a round-robin fashion. To minimize seek overhead, the block size is large (typically 256k, but can be configured between 16k and 1M). Large blocks give the same advantage as extents in file systems such as Veritas [15]: they allow a large amount of data to be retrieved in a single I/O from each disk. GPFS stores small files (and the end of large files) in smaller units called sub-blocks, which are as small as 1/32 of the size of a full block.

To exploit disk parallelism when reading a large file from a single-threaded application GPFS prefetches data into its buffer pool, issuing I/O requests in parallel to as many disks as necessary to achieve the bandwidth of which the switching fabric is capable. Similarly, dirty data buffers that are no longer being accessed are written to disk in parallel. This approach allows reading or writing data from/to a single file at the aggregate data rate supported by the underlying disk subsystem and interconnection fabric. GPFS recognizes sequential, reverse sequential, as well as various forms of strided access patterns. For applications that do not fit one of these patterns, GPFS provides an interface that allows passing prefetch hints to the file system [13].

Striping works best when disks have equal size and performance. A non-uniform disk configuration requires a trade-off between throughput and space utilization: maximizing space utilization means placing more data on larger disks, but this reduces total throughput, because larger disks will then receive a proportionally larger fraction of I/O requests, leaving

the smaller disks under-utilized. GPFS allows the administrator to make this trade-off by specifying whether to balance data placement for throughput or space utilization.

2.2 Large Directory Support

To support efficient file name lookup in very large directories (millions of files), GPFS uses *extensible hashing* [6] to organize directory entries within a directory. For directories that occupy more than one disk block, the block containing the directory entry for a particular name can be found by applying a hash function to the name and using the n low-order bits of the hash value as the block number, where n depends on the size of the directory.

As a directory grows, extensible hashing adds new directory blocks one at a time. When a create operation finds no more room in the directory block designated by the hash value of the new name, it splits the block in two. The logical block number of the new directory block is derived from the old block number by adding a '1' in the $n+1^{st}$ bit position, and directory entries with a '1' in the $n+1^{st}$ bit of their hash value are moved to the new block. Other directory blocks remain unchanged. A large directory is therefore, in general, represented as a sparse file, with holes in the file representing directory blocks that have not yet been split. By checking for sparse regions in the directory file, GPFS can determine how often a directory block has been split, and thus how many bits of the hash value to use to locate the directory block containing a given name. Hence a lookup always requires only a single directory block access, regardless of the size and structure of the directory file [7].

2.3 Logging and Recovery

In a large file system it is not feasible to run a file system check (fsck) to verify/restore file system consistency each time the file system is mounted or every time that one of the nodes in a cluster goes down. Instead, GPFS records all metadata updates that affect file system consistency in a journal or write-ahead log [8]. User data are not logged.

Each node has a separate log for each file system it mounts, stored in that file system. Because this log can be read by all other nodes, any node can perform recovery on behalf of a failed node – it is not necessary to wait for the failed node to come back to life. After a failure, file system consistency is restored quickly by simply re-applying all updates recorded in the failed node's log.

For example, creating a new file requires updating a directory block as well as the inode of the new file. After acquiring locks on the directory block and the inode, both are updated in the buffer cache, and log records are spooled that describe both updates. Before the modified inode or directory block are allowed to be written back to disk, the corresponding log records must be forced to disk. Thus, for example, if the node fails after writing the directory block but before the inode is written to disk, the node's log is guaranteed to contain the log record that is necessary to redo the missing inode update.

Once the updates described by a log record have been written back to disk, the log record is no longer needed and can be discarded. Thus, logs can be fixed size, because space in the log can be freed up at any time by flushing dirty metadata back to disk in the background.

3 Managing Parallelism and Consistency in a Cluster

3.1 Distributed Locking vs. Centralized Management

A cluster file system allows scaling I/O throughput beyond what a single node can achieve. To exploit this capability requires reading and writing in parallel from all nodes in the cluster. On the other hand, preserving file system consistency and POSIX semantics requires synchronizing access to data and metadata from multiple nodes, which potentially limits parallelism. GPFS guarantees single-node equivalent POSIX semantics for file system operations across the cluster. For example if two processes on different nodes access the same file, a read on one node will see either all or none of the data written by a concurrent write operation on the other node (read/write atomicity). The only exceptions are access time updates, which are not immediately visible on all nodes.[1]

There are two approaches to achieving the necessary synchronization:

1. *Distributed Locking*: every file system operation acquires an appropriate read or write lock to synchronize with conflicting operations on other nodes before reading or updating any file system data or metadata.

[1] Since read-read sharing is very common, synchronizing atime across multiple nodes would be prohibitively expensive. Since there are few, if any, applications that require accurate atime, we chose to propagate atime updates only periodically.

2. *Centralized Management*: all conflicting operations are forwarded to a designated node, which performs the requested read or update.

The GPFS architecture is fundamentally based on distributed locking. Distributed locking allows greater parallelism than centralized management as long as different nodes operate on different pieces of data/metadata. On the other hand, data or metadata that is frequently accessed and updated from different nodes may be better managed by a more centralized approach: when lock conflicts are frequent, the overhead for distributed locking may exceed the cost of forwarding requests to a central node. Lock granularity also impacts performance: a smaller granularity means more overhead due to more frequent lock requests, whereas a larger granularity may cause more frequent lock conflicts.

To efficiently support a wide range of applications no single approach is sufficient. Access characteristics vary with workload and are different for different types of data, such as user data vs. file metadata (e.g., modified time) vs. file system metadata (e.g., allocation maps). Consequently, GPFS employs a variety of techniques to manage different kinds of data: byte-range locking for updates to user data, dynamically elected "metanodes" for centralized management of file metadata, distributed locking with centralized hints for disk space allocation, and a central coordinator for managing configuration changes.

In the following sections we first describe the GPFS distributed lock manager and then discuss how each of the techniques listed above improve scalability by optimizing – or in some cases avoiding – the use of distributed locking.

3.2 The GPFS Distributed Lock Manager

The GPFS distributed lock manager, like many others [2, 9], uses a centralized global lock manager running on one of the nodes in the cluster, in conjunction with local lock managers in each file system node. The global lock manager coordinates locks between local lock managers by handing out *lock tokens* [9], which convey the right to grant distributed locks without the need for a separate message exchange each time a lock is acquired or released. Repeated accesses to the same disk object from the same node only require a single message to obtain the right to acquire a lock on the object (the lock token). Once a node has obtained the token from the global lock manager (also referred as the *token manager* or *token server*), subsequent operations issued on the same node can acquire a lock on the same object without requiring additional messages. Only when an operation on another node requires a conflicting lock on the same object are additional messages necessary to revoke the lock token from the first node so it can be granted to the other node.

Lock tokens also play a role in maintaining cache consistency between nodes. A token allows a node to cache data it has read from disk, because the data cannot be modified elsewhere without revoking the token first.

3.3 Parallel Data Access

Certain classes of supercomputer applications require writing to the same file from multiple nodes. GPFS uses byte-range locking to synchronize reads and writes to file data. This approach allows parallel applications to write concurrently to different parts of the same file, while maintaining POSIX read/write atomicity semantics. However, were byte-range locks implemented in a naive manner, acquiring a token for a byte range for the duration of the read/write call and releasing it afterwards, locking overhead would be unacceptable. Therefore, GPFS uses a more sophisticated byte-range locking protocol that radically reduces lock traffic for many common access patterns.

Byte-range tokens are negotiated as follows. The first node to write to a file will acquire a byte-range token for the whole file (zero to infinity). As long as no other nodes access the same file, all read and write operations are processed locally without further interactions between nodes. When a second node begins writing to the same file it will need to revoke at least part of the byte-range token held by the first node. When the first node receives the revoke request, it checks whether the file is still in use. If the file has since been closed, the first node will give up the whole token, and the second node will then be able to acquire a token covering the whole file. Thus, in the absence of concurrent write sharing, byte-range locking in GPFS behaves just like whole-file locking and is just as efficient, because a single token exchange is sufficient to access the whole file.

On the other hand, if the second node starts writing to a file *before* the first node closes the file, the first node will relinquish only part of its byte-range token. If the first node is writing sequentially at offset o_1 and the second node at offset o_2, the first node will relinquish its token from o_2 to infinity (if $o_2 > o_1$) or from zero to o_1 (if $o_2 < o_1$). This will allow both nodes to continue writing forward from their current write offsets without further token conflicts. In general, when multiple nodes

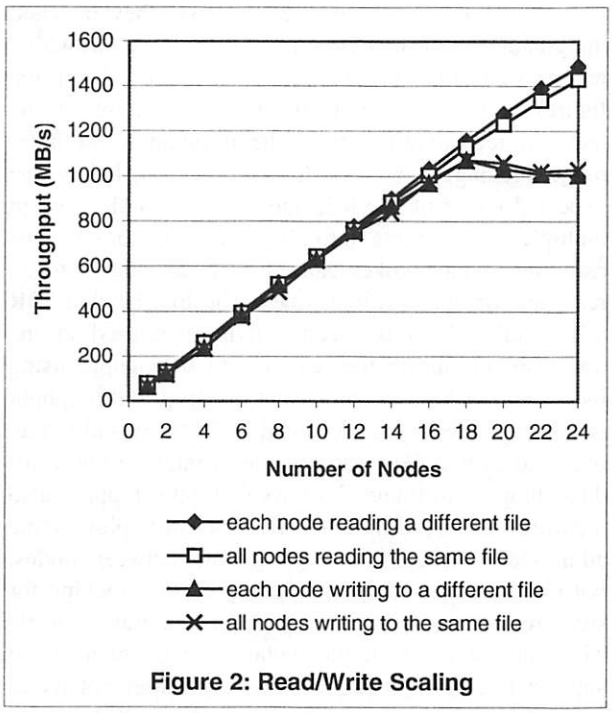


Figure 2: Read/Write Scaling

are writing sequentially to non-overlapping sections of the same file, each node will be able to acquire the necessary token with a single token exchange as part of its first write operation.

Information about write offsets is communicated during this token negotiation by specifying a *required range*, which corresponds to the offset and length of the write() system call currently being processed, and a *desired range*, which includes likely future accesses. For sequential access, the desired range will be from the current write offset to infinity. The token protocol will revoke byte ranges only from nodes that conflict with the required range; the token server will then grant as large a sub-range of the desired range as is possible without conflicting with ranges still held by other nodes.

The measurements shown in Figure 2 demonstrate how I/O throughput in GPFS scales when adding more file system nodes and more disks to the system. The measurements were obtained on a 32-node IBM RS/6000 SP system with 480 disks configured as 96 4+P RAID-5 devices attached to the SP switch through two I/O server nodes. The figure compares reading and writing a single large file from multiple nodes in parallel against each node reading or writing a different file. In the single-file test, the file was partitioned into *n* large, contiguous sections, one per node, and each node was reading or writing sequentially to one of the sections. The writes were updates in place to an existing file. The graph starts with a single file system node using four RAIDs on the left, adding four more RAIDs for each node added to the test, ending with 24 file system nodes using all 96 RAIDs on the right. It shows nearly linear scaling in all tested configurations for reads. In the test system, the data throughput was not limited by the disks, but by the RAID controller. The read throughput achieved by GPFS matched the throughput of raw disk reads through the I/O subsystem. The write throughput showed similar scalability. At 18 nodes the write throughput leveled off due to a problem in the switch adapter microcode.[2] The other point to note in this figure is that writing to a single file from multiple nodes in GPFS was just as fast as each node writing to a different file, demonstrating the effectiveness of the byte-range token protocol described above.

As long as the access pattern allows predicting the region of the file being accessed by a particular node in the near future, the token negotiation protocol will be able to minimize conflicts by carving up byte-range tokens among nodes accordingly. This applies not only to simple sequential access, but also to reverse sequential and forward or backward strided access patterns, provided each node operates in different, relatively large regions of the file (coarse-grain sharing).

As sharing becomes finer grain (each node writing to multiple, smaller regions), the token state and corresponding message traffic will grow. Note that byte-range tokens not only guarantee POSIX semantics but also synchronize I/O to the data blocks of the file. Since the smallest unit of I/O is one sector, the byte-range token granularity can be no smaller than one sector; otherwise, two nodes could write to the same sector at the same time, causing lost updates. In fact, GPFS uses byte-range tokens to synchronize data block allocation as well (see next section), and therefore rounds byte-range tokens to block boundaries. Hence multiple nodes writing into the same data block will cause token conflicts even if individual write operations do not overlap ("false sharing").

To optimize fine-grain sharing for applications that do not require POSIX semantics, GPFS allows disabling normal byte-range locking by switching to *data shipping* mode. File access switches to a method best

[2] The test machine had pre-release versions of the RS/6000 SP Switch2 adapters. In this early version of the adapter microcode, sending data from many nodes to a single I/O server node was not as efficient as sending from an I/O server node to many nodes.

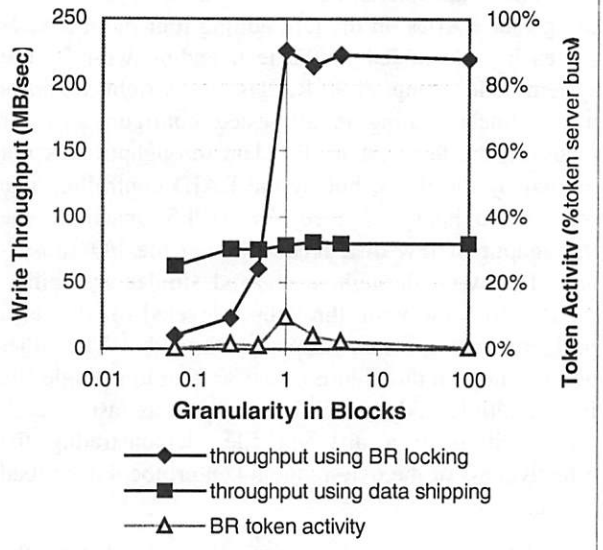


Figure 3: Effect of Sharing Granularity on Write Throughput

described as partitioned, centralized management. File blocks are assigned to nodes in a round-robin fashion, so that each data block will be read or written only by one particular node. GPFS forwards read and write operations originating from other nodes to the node responsible for a particular data block. For fine-grain sharing this is more efficient than distributed locking, because it requires fewer messages than a token exchange, and it avoids the overhead of flushing dirty data to disk when revoking a token. The eventual flushing of data blocks to disk is still done in parallel, since the data blocks of the file are partitioned among many nodes.

Figure 3 shows the effect of sharing granularity on write throughput, using data shipping and using normal byte-range locking. These measurements were done on a smaller SP system with eight file system nodes and two I/O servers, each with eight disks. Total throughput to each I/O server was limited by the switch.[3] We measured throughput with eight nodes updating fixed size records within the same file. The test used a strided access pattern, where the first node wrote records 0, 8, 16, ..., the second node wrote records 1, 9, 17, ..., and so on. The larger record sizes (right half of the figure) were multiples of the file system block size. Figure 3 shows that byte-range locking achieved nearly the full I/O throughput for these sizes, because they matched the granularity of the byte-range tokens. Updates with a record size smaller than one block (left half of the figure) required twice as much I/O, because of the required read-modify-write. The throughput for byte-range locking, however, dropped off far below the expected factor of one half due to token conflicts when multiple nodes wrote into the same data block. The resulting token revokes caused each data block to be read and written multiple times. The line labeled "BR token activity" plots token activity measured at the token server during the test of I/O throughput using byte-range locking. It shows that the drop in throughput was in fact due to the additional I/O activity and not an overload of the token server. The throughput curve for data shipping in Figure 3 shows that data shipping also incurred the read-modify-write penalty plus some additional overhead for sending data between nodes, but it dramatically outperformed byte-range locking for small record sizes that correspond to fine grain sharing. The data shipping implementation was intended to support fine-grain access, and as such does not try to avoid the read-modify-write for record sizes larger than a block. This fact explains why data shipping throughput stayed flat for larger record sizes.

Data shipping is primarily used by the MPI/IO library. MPI/IO does not require POSIX semantics, and provides a natural mechanism to define the collective that assigns blocks to nodes. The programming interfaces used by MPI/IO to control data shipping, however, are also available to other applications that desire this type of file access [13].

3.4 Synchronizing Access to File Metadata

Like other file systems, GPFS uses inodes and indirect blocks to store file attributes and data block addresses. Multiple nodes writing to the same file will result in concurrent updates to the inode and indirect blocks of the file to change file size and modification time (mtime) and to store the addresses of newly allocated data blocks. Synchronizing updates to the metadata on disk via exclusive write locks on the inode would result in a lock conflict on every write operation.

Instead, write operations in GPFS use a *shared write lock* on the inode that allows concurrent writers on multiple nodes. This shared write lock only conflicts with operations that require exact file size and/or mtime (a stat() system call or a read operation that attempts to read past end-of-file). One of the nodes accessing the file is designated as the *metanode* for the file; only the metanode reads or writes the inode from or to disk. Each writer updates a locally cached copy of the inode

[3] Here, the older RS/6000 SP Switch, with nominal 150 MB/sec throughput. Software overhead reduces this to approximately 125 MB/sec.

and forwards its inode updates to the metanode periodically or when the shared write token is revoked by a stat() or read() operation on another node. The metanode merges inode updates from multiple nodes by retaining the largest file size and latest mtime values it receives. Operations that update file size or mtime non-monotonically (trunc() or utimes()) require an exclusive inode lock.

Updates to indirect blocks are synchronized in a similar fashion. When writing a new file, each node independently allocates disk space for the data blocks it writes. The synchronization provided by byte-range tokens ensures that only one node will allocate storage for any particular data block. This is the reason that GPFS rounds byte-range tokens to block boundaries. Periodically or on revocation of a byte-range token, the new data block addresses are sent to the metanode, which then updates the cached indirect blocks accordingly.

Thus, GPFS uses distributed locking to guarantee POSIX semantics (e.g., a stat() system call sees the file size and mtime of the most recently completed write operation), but I/O to the inode and indirect blocks on disk is synchronized using a centralized approach (forwarding inode updates to the metanode). This allows multiple nodes to write to the same file without lock conflicts on metadata updates and without requiring messages to the metanode on every write operation.

The metanode for a particular file is elected dynamically with the help of the token server. When a node first accesses a file, it tries to acquire the *metanode token* for the file. The token is granted to the first node to do so; other nodes instead learn the identity of the metanode. Thus, in traditional workloads without concurrent file sharing, each node becomes metanode for the files it uses and handles all metadata updates locally.

When a file is no longer being accessed on the metanode and ages out of the cache on that node, the node relinquishes its metanode token and stops acting as metanode. When it subsequently receives a metadata request from another node, it sends a negative reply; the other node will then attempt to take over as metanode by acquiring the metanode token. Thus, the metanode for a file tends to stay within the set of nodes actively accessing that file.

3.5 Allocation Maps

The allocation map records the allocation status (free or in-use) of all disk blocks in the file system. Since each disk block can be divided into up to 32 subblocks to store data for small files, the allocation map contains 32 bits per disk block as well as linked lists for finding a free disk block or a subblock of a particular size efficiently.

Allocating disk space requires updates to the allocation map, which must be synchronized between nodes. For proper striping, a write operation must allocate space for a particular data block on a particular disk, but given the large block size used by GPFS, it is not as important where on that disk the data block is written. This fact allows organizing the allocation map in a way that minimizes conflicts between nodes by interleaving free space information about different disks in the allocation map as follows. The map is divided into a large, fixed number n of separately lockable *regions*, and each region contains the allocation status of $1/n^{th}$ of the disk blocks on every disk in the file system. This map layout allows GPFS to allocate disk space properly striped across all disks by accessing only a single allocation region at a time. This approach minimizes lock conflicts, because different nodes can allocate space from different regions. The total number of regions is determined at file system creation time based on the expected number of nodes in the cluster.

For each GPFS file system, one of the nodes in the cluster is responsible for maintaining free space statistics about all allocation regions. This *allocation manager* node initializes free space statistics by reading the allocation map when the file system is mounted. The statistics are kept loosely up-to-date via periodic messages in which each node reports the net amount of disk space allocated or freed during the last period. Instead of all nodes individually searching for regions that still contain free space, nodes ask the allocation manager for a region to try whenever a node runs out of disk space in the region it is currently using. To the extent possible, the allocation manager prevents lock conflicts between nodes by directing different nodes to different regions.

Deleting a file also updates the allocation map. A file created by a parallel program running on several hundred nodes might have allocated blocks in several hundred regions. Deleting the file requires locking and updating each of these regions, perhaps stealing them from the nodes currently allocating out of them, which could have a disastrous impact on performance.

Therefore, instead of processing all allocation map updates at the node on which the file was deleted, those that update regions known to be in use by other nodes are sent to those nodes for execution. The allocation manager periodically distributes hints about which

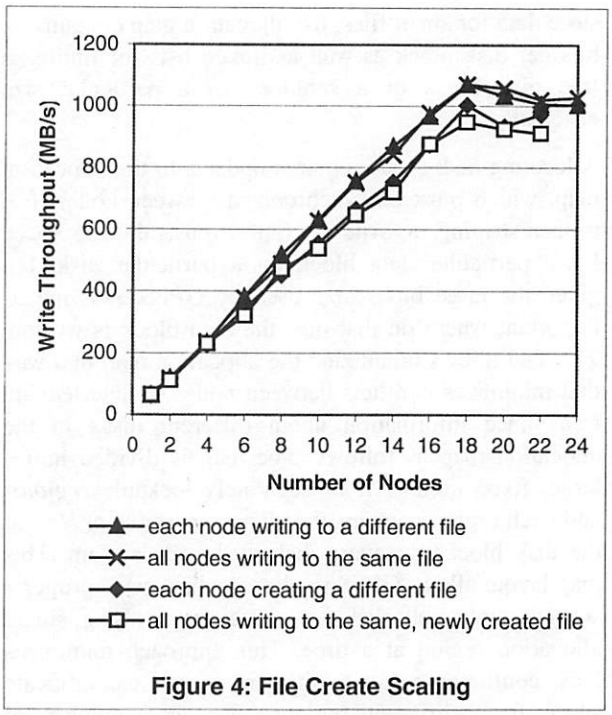


Figure 4: File Create Scaling

regions are in use by which nodes to facilitate shipping deallocation requests.

To demonstrate the effectiveness of the allocation manager hints as well as the metanode algorithms described in the previous section we measured write throughput for updates in place to an existing file against creation of a new file (Figure 4). As in Figure 2, we measured all nodes writing to a single file (using the same access pattern as in Figure 2), as well as each node writing to a different file. The measurements were done on the same hardware as Figure 2, and the data points for the write throughput are in fact the same points shown in the earlier figure. Due to the extra work required to allocate disk storage, throughput for file creation was slightly lower than for update in place. Figure 4 shows, however, that create throughput still scaled nearly linearly with the number of nodes, and that creating a single file from multiple nodes was just as fast as each node creating a different file.

3.6 Other File System Metadata

A GPFS file system contains other global metadata, including file system configuration data, space usage quotas, access control lists, and extended attributes. Space does not permit a detailed description of how each of these types of metadata is managed, but a brief mention is in order. As in the cases described in previous sections, GPFS uses distributed locking to protect the consistency of the metadata on disk, but in most cases uses more centralized management to coordinate or collect metadata updates from different nodes. For example, a quota manager hands out relatively large increments of disk space to the individual nodes writing a file, so quota checking is done locally, with only occasional interaction with the quota manager.

3.7 Token Manager Scaling

The token manager keeps track of all lock tokens granted to all nodes in the cluster. Acquiring, relinquishing, upgrading, or downgrading a token requires a message to the token manager. One might reasonably expect that the token manager could become a bottleneck in a large cluster, or that the size of the token state might exceed the token manager's memory capacity.

One way to address these issues would be to partition the token space and distribute the token state among several nodes in the cluster. We found, however, that this is not the best way — or at least not the most important way — to address token manager scaling issues, for the following reasons.

A straightforward way to distribute token state among nodes might be to hash on the file inode number. Unfortunately, this does not address the scaling issues arising from parallel access to a single file. In the worst case, concurrent updates to a file from multiple nodes generate a byte-range token for each data block of a file. Because the size of a file is effectively unbounded, the size of the byte-range token state for a single file is also unbounded. One could conceivably partition byte-range token management for a single file among multiple nodes, but this would make the frequent case of a single node acquiring a token for a whole file prohibitively expensive. Instead, the token manager prevents unbounded growth of its token state by monitoring its memory usage and, if necessary, revoking tokens to reduce the size of the token state[4].

The most likely reason for a high load on the token manager node is lock conflicts that cause token revocation. When a node downgrades or relinquishes a token, dirty data or metadata covered by that token must be flushed to disk and/or discarded from the cache. As explained earlier (see Figure 3), the cost of disk I/O caused by token conflicts dominates the cost of token manager messages. Therefore, a much more

[4] Applications that do not require POSIX semantics can, of course, use data shipping to bypass byte-range locking and avoid token state issues.

effective way to reduce token manager load and improve overall performance is to avoid lock conflicts in the first place. The allocation manager hints described in Section 3.5 are an example of avoiding lock conflicts.

Finally, GPFS uses a number of optimizations in the token protocol that significantly reduce the cost of token management and improve response time as well. When it is necessary to revoke a token, it is the responsibility of the revoking node to send revoke messages to all nodes that are holding the token in a conflicting mode, to collect replies from these nodes, and to forward these as a single message to the token manager. Acquiring a token will never require more than two messages to the token manager, regardless of how many nodes may be holding the token in a conflicting mode.

The protocol also supports token prefetch and token request batching, which allow acquiring multiple tokens in a single message to the token manager. For example, when a file is accessed for the first time, the necessary inode token, metanode token, and byte-range token to read or write the file are acquired with a single token manager request.

When a file is deleted on a node, the node does not immediately relinquish the tokens associated with that file. The next file created by the node can then re-use the old inode and will not need to acquire any new tokens. A workload where users on different nodes create and delete files under their respective home directories will generate little or no token traffic.

Figure 5 demonstrates the effectiveness of this optimization. It shows token activity while running a multi-user file server workload on multiple nodes. The workload was generated by the dbench program [10], which simulates the file system activity of the NetBench [11] file server benchmark. It ran on eight file system nodes, with each node running a workload for 10 NetBench clients. Each client ran in a different subdirectory. The figure shows an initial spike of token activity as the benchmark started up on all of the nodes and each node acquired tokens for the files accessed by its clients. Even though the benchmark created and deleted many files throughout the run, each node reused a limited number of inodes. Once all nodes had obtained a sufficient number of inodes, token activity quickly dropped to near zero. Measurements of the CPU load on the token server indicated that it is capable of supporting between 4000 and 5000 token requests per second, so the peak request rate shown in Figure 5 consumed only a small fraction of the token server capacity. Even the height of the peak was only an artifact of starting the benchmark at the same time on all of the nodes, which would not be likely to happen under a real multi-user workload.

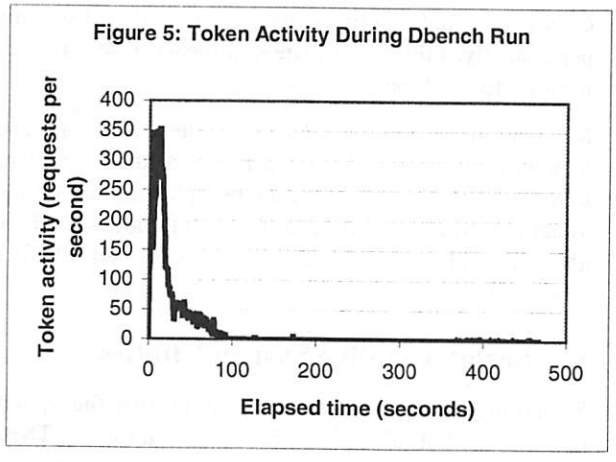


Figure 5: Token Activity During Dbench Run

4 Fault Tolerance

As a cluster is scaled up to large numbers of nodes and disks it becomes increasingly unlikely that all components are working correctly at all times. This implies the need to handle component failures gracefully and continue operating in the presence of failures.

4.1 Node Failures

When a node fails, GPFS must restore metadata being updated by the failed node to a consistent state, must release resources held by the failed node (lock tokens), and it must appoint replacements for any special roles played by the failed node (e.g., metanode, allocation manager, or token manager).

Since GPFS stores recovery logs on shared disks, metadata inconsistencies due to a node failure are quickly repaired by running log recovery from the failed node's log on one of the surviving nodes. After log recovery is complete, the token manager releases tokens held by the failed node. The distributed locking protocol ensures that the failed node must have held tokens for all metadata it had updated in its cache but had not yet written back to disk at the time of the failure. Since these tokens are only released after log recovery is complete, metadata modified by the failed node will not be accessible to other nodes until it is known to be in a consistent state. This observation is true even in cases where GPFS uses a more centralized approach to synchronizing metadata updates, for example, the file size and mtime updates that are collected by the metanode. Even though the write operations causing such updates are not synchronized

via distributed locking, the updates to the metadata on disk are still protected by the distributed locking protocol — in this case, through the metanode token.

After log recovery completes, other nodes can acquire any metanode tokens that had been held by the failed node and thus take over the role of metanode. If another node had sent metadata updates to the old metanode but, at the time of the failure, had not yet received an acknowledgement that the updates were committed to disk, it re-sends the updates to the new metanode. These updates are idempotent, so the new metanode can simply re-apply them.

Should the token manager fail, another node will take over this responsibility and reconstruct the token manager state by querying all surviving nodes about the tokens they currently hold. Since the new token manager does not know what tokens were held by failed nodes, it will not grant any new tokens until log recovery is complete. Tokens currently held by the surviving nodes are not affected by this.

Similarly, other special functions carried out by a failed node (e.g., allocation manager) are assigned to another node, which rebuilds the necessary state by reading information from disk and/or querying other nodes.

4.2 Communication Failures

To detect node failures GPFS relies on a group services layer that monitors nodes and communication links via periodic heartbeat messages and implements a process group membership protocol [12]. When a node fails, the group services layer informs the remaining nodes of a group membership change. This triggers the recovery actions described in the previous section.

A communication failure such as a bad network adapter or a loose cable may cause a node to become isolated from the others, or a failure in the switching fabric may cause a network partition. Such a partition is indistinguishable from a failure of the unreachable nodes. Nodes in different partitions may still have access to the shared disks and would corrupt the file system if they were allowed to continue operating independently of each other. For this reason, GPFS allows accessing a file system only by the group containing a majority of the nodes in the cluster; the nodes in the minority group will stop accessing any GPFS disk until they can re-join a majority group.

Unfortunately, the membership protocol cannot guarantee how long it will take for each node to receive and process a failure notification. When a network partition occurs, it is not known when the nodes that are no longer members of the majority will be notified and stop accessing shared disks. Therefore, before starting log recovery in the majority group, GPFS *fences* nodes that are no longer members of the group from accessing the shared disks, i.e., it invokes primitives available in the disk subsystem to stop accepting I/O requests from the other nodes.

To allow fault-tolerant two-node configurations, a communication partition in such a configuration is resolved exclusively through disk fencing instead of using a majority rule: upon notification of the failure of the other node, each node will attempt to fence all disks from the other node in a predetermined order. In case of a network partition, only one of the two nodes will be successful and can continue accessing GPFS file systems.

4.3 Disk Failures

Since GPFS stripes data and metadata across all disks that belong to a file system, the loss of a single disk will affect a disproportionately large fraction of the files. Therefore, typical GPFS configurations use dual-attached RAID controllers, which are able to mask the failure of a physical disk or the loss of an access path to a disk. Large GPFS file systems are striped across multiple RAID devices. In such configurations, it is important to match the file system block size and alignment with RAID stripes so that data block writes do not incur a write penalty for the parity update.

As an alternative or a supplement to RAID, GPFS supports replication, which is implemented in the file system. When enabled, GPFS allocates space for two copies of each data or metadata block on two different disks and writes them to both locations. When a disk becomes unavailable, GPFS keeps track of which files had updates to a block with a replica on the unavailable disk. If and when the disk becomes available again, GPFS brings stale data on the disk up-to-date by copying the data from another replica. If a disk fails permanently, GPFS can instead allocate a new replica for all affected blocks on other disks.

Replication can be enabled separately for data and metadata. In cases where part of a disk becomes unreadable (bad blocks), metadata replication in the file system ensures that only a few data blocks will be affected, rather than rendering a whole set of files inaccessible.

5 Scalable Online System Utilities

Scalability is important not only for normal file system operations, but also for file system utilities. These

utilities manipulate significant fractions of the data and metadata in the file system, and therefore benefit from parallelism as much as parallel applications.

For example, GPFS allows growing, shrinking, or reorganizing a file system by adding, deleting, or replacing disks in an existing file system. After adding new disks, GPFS allows rebalancing the file system by moving some of the existing data to the new disks. To remove or replace one or more disks in a file system, GPFS must move all data and metadata off the affected disks. All of these operations require reading all inodes and indirect blocks to find the data that must be moved. Other utilities that need to read all inodes and indirect blocks include defragmentation, quota-check, and fsck[5] [13]. Also, when replication is enabled and a group of disks that were down becomes available again, GPFS must perform a metadata scan to find files with missing updates that need to be applied to these disks.

Finishing such operations in any reasonable amount of time requires exploiting the parallelism available in the system. To this end, GPFS appoints one of the nodes as a *file system manager* for each file system, which is responsible for coordinating such administrative activity. The file system manager hands out a small range of inode numbers to each node in the cluster. Each node processes the files in its assigned range and then sends a message to the file system manager requesting more work. Thus, all nodes work on different subsets of files in parallel, until all files have been processed. During this process, additional messages may be exchanged between nodes to compile global file system state. While running fsck, for example, each node collects block references for a different section of the allocation map. This allows detecting inter-file inconsistencies, such as a single block being assigned to two different files.

Even with maximum parallelism, these operations can take a significant amount of time on a large file system. For example, a complete rebalancing of a multi-terabyte file system may take several hours. Since it is unacceptable for a file system to be unavailable for such a long time, GPFS allows all of its file system utilities to run on-line, i.e., while the file system is mounted and accessible to applications. The only exception is a full file system check for diagnostic purposes (fsck), which requires the file system to be unmounted. File system utilities use normal distributed locking to synchronize with other file activity. Special synchronization is required only while reorganizing higher-level metadata (allocation maps and the inode file). For example, when it is necessary to move a block of inodes from one disk to another, the node doing so acquires a special range lock on the inode file, which is more efficient and less disruptive than acquiring individual locks on all inodes within the block.

6 Experiences

GPFS is installed at several hundred customer sites, on clusters ranging from a few nodes with less than a terabyte of disk, up to the 512-node ASCI White system with its 140 terabytes of disk space in two file systems. Much has been learned that has affected the design of GPFS as it has evolved. These lessons would make a paper in themselves, but a few are of sufficient interest to warrant relating here.

Several of our experiences pointed out the importance of intra-node as well as inter-node parallelism and for properly balancing load across nodes in a cluster. In the initial design of the system management commands, we assumed that distributing work by starting a thread on each node in the cluster would be sufficient to exploit all available disk bandwidth. When rebalancing a file system, for example, the strategy of handing out ranges of inodes to one thread per node, as described in Section 5, was able to generate enough I/O requests to keep all disks busy. We found, however, that we had greatly underestimated the amount of skew this strategy would encounter. Frequently, a node would be handed an inode range containing significantly more files or larger files than other ranges. Long after other nodes had finished their work, this node was still at work, running single-threaded, issuing one I/O at a time.

The obvious lesson is that the granules of work handed out must be sufficiently small and of approximately equal size (ranges of blocks in a file rather than entire files). The other lesson, less obvious, is that even on a large cluster, intra-node parallelism is often a more efficient road to performance than inter-node parallelism. On modern systems, which are often high-degree SMPs with high I/O bandwidth, relatively few nodes can saturate the disk system[6]. Exploiting the available bandwidth by running multiple threads per node, for example, greatly reduces the effect of workload skew.

Another important lesson is that even though the small amount of CPU consumed by GPFS centralized

[5] For diagnostic purposes to verify file system consistency, not part of a normal mount.

[6] Sixteen nodes drive the entire 12 GB/s I/O bandwidth of ASCI White.

management functions (e.g., the token manager) normally does not affect application performance, it can have a significant impact on highly parallel applications. Such applications often run in phases with barrier synchronization points. As described in Section 5, centralized services are provided by the file system manager, which is dynamically chosen from the nodes in the cluster. If this node also runs part of the parallel application, the management overhead will cause it to take longer to reach its barrier, leaving the other nodes idle. If the overhead slows down the application *by even one percent*, the idle time incurred by other nodes will be equivalent to leaving five nodes unused on the 512-node ASCI White system! To avoid this problem, GPFS allows restricting management functions to a designated set of administrative nodes. A large cluster can dedicate one or two administrative nodes, avoid running load sensitive, parallel applications on them, and actually increase the available computing resource.

Early versions of GPFS had serious performance problems with programs like "ls –l" and incremental backup, which call stat() for each file in a directory. The stat() call reads the file's inode, which requires a read token. If another node holds this token, releasing it may require dirty data to be written back on that node, so obtaining the token can be expensive. We solved this problem by exploiting parallelism. When GPFS detects multiple accesses to inodes in the same directory, it uses multiple threads to prefetch inodes for other files in the same directory. Inode prefetch speeds up directory scans by almost a factor of ten.

Another lesson we learned on large systems is that even the rarest failures, such as data loss in a RAID, will occur. One particularly large GPFS system experienced a microcode failure in a RAID controller caused by an intermittent problem during replacement of a disk. The failure rendered three sectors of an allocation map unusable. Unfortunately, an attempt to allocate out of one of these sectors generated an I/O error, which caused the file system to take itself off line. Running log recovery repeated the attempt to write the sector and the I/O error. Luckily no user data was lost, and the customer had enough free space in other file systems to allow the broken 14 TB file system to be mounted read-only and copied elsewhere. Nevertheless, many large file systems now use GPFS metadata replication *in addition* to RAID to provide an extra measure of security against dual failures.

Even more insidious than rare, random failures are systematic ones. One customer was unfortunate enough to receive several hundred disk drives from a bad batch with an unexpectedly high failure rate. The customer wanted to replace the bad drives without taking the system down. One might think this could be done by successively replacing each drive and letting the RAID rebuild, but this would have greatly increased the possibility of a dual failure (i.e., a second drive failure in a rebuilding RAID parity group) and a consequent catastrophic loss of the file system. The customer chose as a solution to delete a small number of disks (here, RAID parity groups) from the file system, during which GPFS rebalances data from the deleted disks onto the remaining disks. Then new disks (parity groups) were created from the new drives and added back to the file system (again rebalancing). This tedious process was repeated until all disks were replaced, without taking the file system down and without compromising its reliability. Lessons include not assuming independent failures in a system design, and the importance of online system management and parallel rebalancing.

7 Related Work

One class of file systems extends the traditional file server architecture to a storage area network (SAN) environment by allowing the file server clients to access data directly from disk through the SAN. Examples of such SAN file systems are IBM/Tivoli SANergy [14], and Veritas SANPoint Direct [15]. These file systems can provide efficient data access for large files, but, unlike GPFS, all metadata updates are still handled by a centralized metadata server, which makes this type of architecture inherently less scalable. SAN file systems typically do not support concurrent write sharing, or sacrifice POSIX semantics to do so. For example, SANergy allows multiple clients to read and write to the same file through a SAN, but provides no consistent view of the data unless explicit fcntl locking calls are added to the application program.

SGI's XFS file system [16] is designed for similar, large-scale, high throughput applications that GPFS excels at. It stores file data in large, variable length extents and relies on an underlying logical volume manager to stripe the data across multiple disks. Unlike GPFS however, XFS is not a cluster file system; it runs on large SMPs. CXFS [17] is a cluster version of XFS that allows multiple nodes to access data on shared disks in an XFS file system. However, only one of the nodes handles all metadata updates, like other SAN file systems mentioned above.

Frangipani [18] is a shared-disk cluster file system that is similar in principle to GPFS. It is based on the same, symmetric architecture, and uses similar logging, locking and recovery algorithms based on write-ahead logging with separate logs for each node stored on

shared disk. Like GPFS, it uses a token-based distributed lock manager. A Frangipani file system resides on a single, large (2^{64} byte) virtual disk provided by Petal [19], which redirects I/O requests to a set of Petal servers and handles physical storage allocation and striping. This layered architecture simplifies metadata management in the file system to some extent. The granularity of disk space allocation (64kB) in Petal, however, is too large and its virtual address space is too small to simply reserve a fixed, contiguous virtual disk area (e.g., 1TB) for each file in a Frangipani file system. Therefore, Frangipani still needs its own allocation maps to manage the virtual disk space provided by Petal. Unlike GPFS, Frangipani is mainly "targeted for environments with program development and engineering workloads". It implements whole-file locking only and therefore does not allow concurrent writes to the same file from multiple nodes.

Another example of a shared-disk cluster file system is the Global File System (GFS) [20], which originated as an open source file system for Linux. The newest version (GFS-4) implements journaling, and uses logging, locking, and recovery algorithms similar to those of GPFS and Frangipani. Locking in GFS is closely tied to physical storage. Earlier versions of GFS [21] required locking to be implemented at the disk device via extensions to the SCSI protocol. Newer versions allow the use of an external distributed lock manager, but still lock individual disk blocks of 4kB or 8kB size. Therefore, accessing large files in GFS entails significantly more locking overhead than the byte-range locks used in GPFS. Similar to Frangipani/Petal, striping in GFS is handled in a "Network Storage Pool" layer; once created, however, the stripe width cannot be changed (it is possible to add a new "sub-pools", but striping is confined to a sub-pool, i.e., GFS will not stripe across sub-pools). Like Frangipani, GFS is geared more towards applications with little or no intra-file sharing.

8 Summary and Conclusions

GPFS was built on many of the ideas that were developed in the academic community over the last several years, particularly distributed locking and recovery technology. To date it has been a matter of conjecture how well these ideas scale. We have had the opportunity to test those limits in the context of a product that runs on the largest systems in existence.

One might question whether distributed locking scales, in particular, whether lock contention for access to shared metadata might become a bottleneck that limits parallelism and scalability. Somewhat to our surprise, we found that distributed locking scales quite well. Nevertheless, several significant changes to conventional file system data structures and locking algorithms yielded big gains in performance, both for parallel access to a single large file and for parallel access to large numbers of small files. We describe a number of techniques that make distributed locking work in a large cluster: byte-range token optimizations, dynamic selection of meta nodes for managing file metadata, segmented allocation maps, and allocation hints.

One might similarly question whether conventional availability technology scales. Obviously there are more components to fail in a large system. Compounding the problem, large clusters are so expensive that their owners demand high availability. Add to this the fact that file systems of tens of terabytes are simply too large to back up and restore. Again, we found the basic technology to be sound. The surprises came in the measures that were necessary to provide data integrity and availability. GPFS replication was implemented because at the time RAID was more expensive than replicated conventional disk. RAID has taken over as its price has come down, but even its high level of integrity is not sufficient to guard against the loss of a hundred terabyte file system.

Existing GPFS installations show that our design is able to scale up to the largest super computers in the world and to provide the necessary fault tolerance and system management functions to manage such large systems. Nevertheless, we expect the continued evolution of technology to demand ever more scalability. The recent interest in Linux clusters with inexpensive PC nodes drives the number of components up still further. The price of storage has decreased to the point that customers are seriously interested in petabyte file systems. This trend makes file system scalability an area of interest for research that will continue for the foreseeable future.

9 Acknowledgements

A large number of people at several IBM locations have contributed to the design and implementation of GPFS over the years. Although space does not allow naming all of them here, the following people have significantly contributed to the work reported here: Jim Wyllie, Dan McNabb, Tom Engelsiepen, Marc Eshel, Carol Hartman, Mike Roberts, Wayne Sawdon, Dave Craft, Brian Dixon, Eugene Johnson, Scott Porter, Bob Curran, Radha Kandadai, Lyle Gayne, Mike Schouten, Dave Shapiro, Kuei-Yu Wang Knop, Irit Loy, Benny Mandler, John Marberg, Itai Nahshon, Sybille Schaller, Boaz Shmueli, Roman Talyanski, and Zvi Yehudai.

References

[1] Roger L. Haskin: *Tiger Shark - a scalable file system for multimedia*, IBM Journal of Research and Development, Volume 42, Number 2, March 1998, pp. 185-197.

[2] C. Mohan, Inderpal Narang: Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. VLDB 1991: 193-207.

[3] *IBM builds world's fastest supercomputer to simulate nuclear testing for U.S. Energy Department*. IBM Press Release, Poughkeepsie, N.Y., June 29, 2000. http://www.ibm.com/servers/eserver/pseries/news/pressreleases/2000/jun/asci_white.html

[4] ASCI White. http://www.rs6000.ibm.com/hardware/largescale/supercomputers/asciwhite/

[5] *ASCI White*. http://www.llnl.gov/asci/platforms/white/home/

[6] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong, *Extendible hashing - a fast access method for dynamic files*, ACM Transactions on Database Systems, New York, NY, Volume 4 Number 3, 1979, pages 315-344.

[7] Frank B. Schmuck, James Christopher Wyllie, and Thomas E. Engelsiepen. *Parallel file system and method with extensible hashing*. US Patent No. 05893086.

[8] J. N. Gray, *Notes on database operating systems*, in *Operating systems, an advanced course*, edited by R. Bayer et. al., Springer-Verlag, Berlin, Germany, 1979, pages 393-400.

[9] Ajay Mohindra and Murthy Devarakonda, *Distributed token management in the Calypso file system*, Proceedings of the IEEE Symposium on Parallel and Distributed Processing, New York, 1994.

[10] Dbench benchmark. Available from ftp://samba.org/pub/tridge/dbench/

[11] NetBench benchmark. http://etestinglabs.com/benchmarks/netbench/netbench.asp

[12] *Group Services Programming Guide and Reference*, RS/6000 Cluster Technology, Document Number SA22-7355-01, Second Edition (April 2000), International Business Machines Corporation, 2455 South Road, Poughkeepsie, NY 12601-5400, USA. Also available from http://www.rs6000.ibm.com/resource/aix_resource/sp_books/pssp/index.html

[13] *IBM General Parallel File System for AIX: Administration and Programming Reference*. Document Number SA22-7452-02, Second Edition (December 2000), International Business Machines Corporation, 2455 South Road, Poughkeepsie, NY 12601-5400, USA. Also available from http://www.rs6000.ibm.com/resource/aix_resource/sp_books/gpfs/index.html

[14] Charlotte Brooks, Ron Henkhaus, Udo Rauch, Daniel Thompson. *A Practical Guide to Tivoli SANergy*. IBM Redbook SG246146, June 2001, available from http://www.redbooks.ibm.com/

[15] *VERITAS SANPoint Direct File Access*. Whitepaper, August 2000. VERITAS Software Corporation, Corporate Headquarters, 1600 Plymouth Street, Mountain View, CA 94043.

[16] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. *Scalability in the XFS File System*, Proceedings of the USENIX 1996 Technical Conference, pages 1-14, San Diego, CA, USA, 1996.

[17] *SGI CXS Clustered File System*, Datasheet, Silicon Graphics, Inc., 1600 Amphitheatre Pkwy. Mountain View, CA 94043.

[18] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. *Frangipani: A Scalable Distributed File System*. In Proceedings of the Symposium on Operating Systems Principles, 1997, pages 224-237.

[19] Edward K. Lee and Chandramohan A. Thekkath. *Petal: Distributed Virtual Disks*, In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1996, pages 84-92.

[20] Kenneth W. Preslan, Andrew P. Barry, Jonathan Brassow, Russell Cattelan, Adam Manthei, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, Matthew O'Keefe, Grant Erickson and Manish Agarwal. *Implementing Journaling in a Linux Shared Disk File System.* Seventeenth IEEE Symposium on Mass Storage Systems, March 2000, pages 351-378.

[21] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. *A 64-bit, Shared Disk File System for Linux.* Sixteenth IEEE Mass Storage Systems Symposium, March 15-18, 1999, San Diego, California, pages 351-378.

Exploiting Inter-File Access Patterns Using Multi-Collective I/O*

Gokhan Memik
EE Dept.
UCLA
memik@ee.ucla.edu

Mahmut Kandemir
CSE Dept.
Penn State
kandemir@cse.psu.edu

Alok Choudhary
ECE Dept.
Northwestern University
choudhar@ece.nwu.edu

Abstract

This paper introduces a new concept called Multi-Collective I/O (MCIO) that extends conventional collective I/O to optimize I/O accesses to multiple arrays simultaneously. In this approach, as in collective I/O, multiple processors co-ordinate to perform I/O on behalf of each other if doing so improves overall I/O time. However, unlike collective I/O, MCIO considers multiple arrays simultaneously; that is, it has a more global view of the overall I/O behavior exhibited by application. This paper shows that determining optimal MCIO access pattern is an NP-complete problem, and proposes two different heuristics for the access pattern detection problem (also called the assignment problem).

Both of the heuristics have been implemented within a runtime library, and tested using a large-scale scientific application. Our preliminary results show that MCIO outperforms collective I/O by as much as 87%. Our runtime library-based implementation can be used by users as well as optimizing compilers. Based on our results, we recommend future library designers for I/O-intensive applications to include MCIO in their suite of optimizations.

1 Introduction

Significant strides made in microprocessor performances in the last decade have increased the importance of optimizing I/O performance. While a highly-optimized I/O platform/hardware is a must for optimal I/O, software optimizations can also play a significant role. This is particularly true for a class of large-scale scientific applications whose access patterns can be analyzed by users/compilers and optimized for the best I/O performance. Previous work has attacked this growing I/O problem at the operating system, runtime libraries, compilers and application levels. One of the major characteristics of most of the previous approaches to I/O is that they attempt to optimize I/O access pattern for a single data structure (e.g., disk-resident array) at a time. While this approach can be useful for some applications, we believe that considering the interactions between different data structures manipulated by the same application opens new opportunities for optimization. For example, considering accesses to multiple arrays manipulated by an application can enable inter-array optimizations such as co-locating arrays in disk space or performing prefetching based on history of array accesses.

This paper introduces a new concept called Multi-Collective I/O (MCIO) that extends conventional collective I/O (CIO) to optimize I/O accesses to multiple arrays simultaneously. In this approach, as in collective I/O, multiple processors co-ordinate to perform I/O on behalf of each other if doing so improves overall I/O time. However, unlike collective I/O, MCIO considers multiple arrays simultaneously; that is, it has a more global view of the overall I/O behavior exhibited by application. This paper shows that determining optimal MCIO access pattern is an NP-complete problem, and proposes two different heuristics for the access pattern detection problem (also

*This research was supported in part by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore National Laboratories, by DOE SCIDAC SDM center and by a grant for NSF, EIA-0103023.

called the assignment problem).

Figure 1 highlights the difference between MCIO and traditional collective I/O. In MCIO, we optimize the accesses from multiple processors to *multiple files* taking into account the inter-file access patterns. In CIO, on the other hand, accesses to a *single file* from different processors are combined to a single and larger I/O request to improve the request time. In MCIO, accesses from several processors to several files are combined to increase the efficiency.

MCIO might be very useful in a number of cases. Many large-scale scientific applications access multiple files in a single run. Several of these applications generate separate files for the simulated values. This forces the application to access data from multiple files to gather the required information. A simple analysis of the code reveals this information, which can be used by the MCIO. MCIO can also be utilized when data for different variables are accessed from a single file. For example, *astro3d* [15], the scientific application we have used in this study, accesses six different variables from a single file. These variables are stored in separate buffers, hence six different I/O calls have to be performed to access the information required by the application. MCIO can be utilized in this case. Note that, accessing different variables from a single file is not a separate problem, it is just a special case for the processor-file configuration in MCIO. Most of the scientific applications fall either in the first (accessing multiple files in a single run) or into the second (performing consecutive I/O calls to a single file to access different variables) category. Therefore, a large majority of the large-scale scientific applications can effectively utilize MCIO.

Another usage of MCIO is related to sub-filing. Memik et al. [16] uses sub-filing for optimizing random accesses to the tape-residing data. In this framework, a large global file is stored as several independent so called *sub-files*. In general, each access to the global file might involve several sub-files. In other words, an access to the global file brings only the smallest subset of sub-files (from tape to disk) that (collectively) contain the required data portion. This not only reduces the effective latency observed from the tape device, but also allows customized management of individual sub-files across storage hierarchy depending on data reuse. MCIO can be utilized to access data that has been brought to the disks.

In this paper, we make the following major contributions:

- We introduce the MCIO technique that improves over the current state-of-the art collective I/O technique;
- We discuss the complexity of the MCIO, and present two different heuristics (one based on sorting and the other based on maximal matching) to perform effective MCIO; and
- We evaluate the effectiveness of the MCIO by using both synthetic workloads and a complete large-scale scientific application.

Both of the heuristics have been implemented within a runtime library, and tested using a large-scale scientific application. Our preliminary results show that MCIO outperforms conventional collective I/O by as much as 87%. Based on our results, we recommend future library designers for I/O-intensive applications to include MCIO in their suite of optimizations.

The remainder of this paper is organized as follows. In the next section, we discuss related work on software-based I/O optimizations. In Section 3, we summarize collective I/O. In Section 4, we discuss the MCIO in detail, and show that finding the optimal access pattern for MCIO is NP-complete. In Section 5, we discuss two different heuristics for determining suitable access patterns. Section 6 introduces our experimental environment and discusses our preliminary results. Finally, we conclude the paper with a summary and an outline of future work in Section 7.

2 Related Work

In collective I/O, individual I/O requests are combined to create a single, big I/O request and sent to the storage system. As a result, the effective I/O bandwidth is

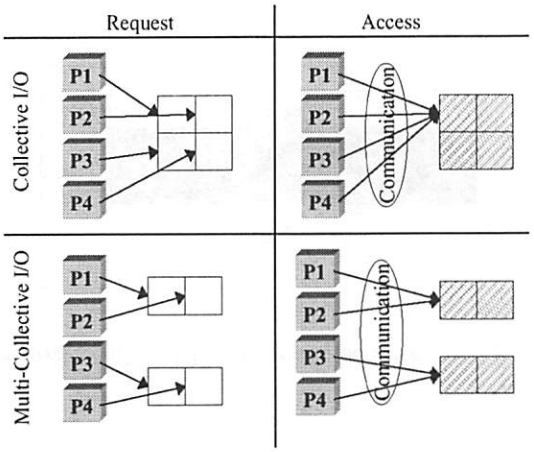


Figure 1: Overview of the MCIO.

significantly increased. This optimization has many variants [19, 13, 22]; although any CIO technique can be used for MCIO, the one used in this study is *two-phase I/O* as implemented in ROMIO, a portable implementation of MPI-IO from Argonne National Laboratories [26]. ROMIO has been incorporated into several MPI-libraries, including the MPI implementations of several vendors (e.g., HP, SGI, NEC) and MPICH and LAM, two widely-used, freely available, portable MPI implementations.

Numerous techniques have been proposed in literature to optimize I/O accesses. The run-time system optimizations [14, 12, 6, 4, 5] are the most relevant among these studies. Although these techniques share the same goal with our work (optimizing I/O accesses at the run-time), they try to optimize only a single file at a time.

Several researchers focused on implementing easy-to-use interfaces that include optimizations for data-intensive scientific applications [24, 3, 22, 23]. These interfaces try to improve the I/O performance with little input from the user. MCIO can be employed by these interfaces with little or no modification.

Characterizing the I/O behavior of the scientific applications has been extensively studied. Cypher et al. [9] studied individual parallel scientific applications, measuring temporal patterns in I/O rates. Crandall et al. [8] performed an analysis based on Pablo [1] on three scientific applications. Nieuwejaar et al. [18] characterized a mix of user programs on Intel iPSC and CM-5. All these studies implicitly motivate the usage of MCIO by illustrating that multiple files are accessed within a single run of the studied application. According to these studies, the number of files accessed varies according to the application and the file system studied but can be as high as 2000.

Several parallel I/O APIs provide routines for accessing multiple files with a single call. For example, HPSS [7] uses the notion of a 'file set' to define a collection of files. The files in a file set can be manipulated as if they constitute a single file. Similarly, SRB [2] uses 'collections' to define a set of files.

3 Collective I/O

Since MCIO is an extended form of conventional collective I/O, in this section we briefly review the fundamental idea behind collective I/O. In many parallel I/O-intensive applications that access large, multidimensional, disk-resident datasets, the performance of I/O accesses depends largely on both the layout of data in files (*storage pattern*) and distribution of data across processors (*access pattern*). In cases where storage and access patterns do not match, allowing each processor to perform I/O independently might cause processors to issue many I/O requests, each for a small amount of consecutive data. Collective I/O can improve the performance in such cases by first reading the dataset in question in a *layout-conforming* (storage layout friendly) manner, and then distributing the data among the processors (using the inter-processor communication network) to obtain the target access pattern. Of course, in this case, the total data access cost should be computed as the sum of I/O cost and communication cost. Since, in I/O-intensive applications, I/O costs in general dominate communication costs, collective I/O might lead to large savings in overall execution times.

Consider an example file access pattern (to a two-dimensional array) depicted in Figure 2. In this figure, the circles correspond to data elements and arrows indicate the storage pattern of the elements. Note that in this case, the storage pattern is row-major whereas the access pattern is column-major. If no collective I/O technique is used, every processor make 8 small requests (each for

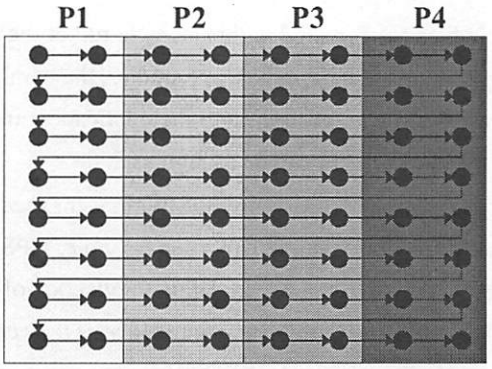


Figure 2: A file access by four processors.

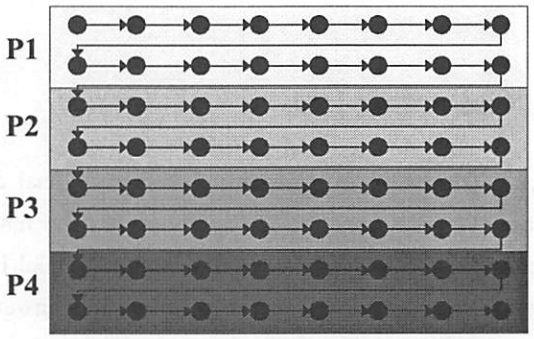


Figure 3: File access using the collective I/O.

two array elements) for a total of 24 small requests to the I/O subsystem (the numbers are for illustrative purposes only). Obviously, this will result in a poor performance as the number of elements read per I/O request is very small. Collective I/O combines these small requests, and sends larger requests to the I/O subsystem to improve the performance. If, for example, the *two-phase I/O* technique is used, in the first step, the processors access the data using a row-major access pattern which is compatible with the (row-major) storage pattern of the array (see Figure 3). Note that this reduces the number of I/O calls as each processor can read as many consecutive data as possible (limited only by available buffer capacity) in a single I/O call. In the second step, the processors engage in all-to-all inter-processor communication so that each processor receives the portion of array it originally requested (that is, each data item is delivered to its final destination).

Several variations of the collective I/O technique have been proposed in previous research. In node grouping [20], nodes making I/O requests are partitioned into groups. Then, they take turns in performing the I/O. In disk-directed I/O [13], one compute node sends the collective request to all I/O nodes. I/O nodes optimize the access, perform the request, and send the data directly back to all compute nodes.

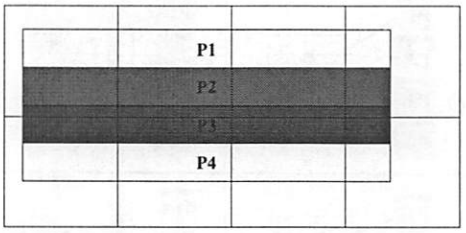


Figure 4: An access pattern involving four processors and eight files.

4 Multi-Collective I/O

In this section, we explain the MCIO technique in detail. Consider Figure 4 where four processors are requesting different portions of eight files. Note that, these files might contain independent data. The problem is to read the corresponding data portions from the corresponding files as fast as possible. There are several methods to achieve this. In a naive method, we would use the CIO for each of the files. For our example in Figure 4, this will result in 8 different I/O calls (each involving two or three processors). In this method, as the number of files increase, the number of I/O calls will increase linearly regardless of the number of processors available. In this paper, we explore methods that improve over this naive method. The main idea behind these methods is to assign different files to different processors, thereby increasing the I/O parallelism available in the system.

In the next subsection, we define the assignment problem of MCIO in detail and show that this problem is NP-complete [11]. There are two important questions that need to be answered. First, how many files should we assign to each processor, or vice versa? Second, once we have determined the number of processors to be assigned for each file, how do we decide which specific processor to assign for each file? In the following subsections, we try to answer these questions. In the rest of this section, we concentrate on the I/O accesses where the number

of files exceeds the number of processors. Note that the problem we try to solve here is symmetric. If the number of files is larger, we try to assign files to processors; otherwise, if the number of processors exceeds the number of files, we try to assign the processors to files. It is easy to see that these two problems are actually duals of each other.

4.1 The Assignment Problem

In this subsection, we first define the assignment problem in detail. Then, we show that the problem of deciding the files to be read by a processor is NP-complete. After this, we build a Linear Programming (LP) model of the problem. As we will show in the following, the assignment variables can either be 0 or 1, hence the model we build is a zero-one integer LP. Since it is slow to construct and solve an LP model in a run-time system, we have also developed two heuristics to solve the problem, which we discuss in the next section.

4.1.1 Definition of the Assignment Problem

In the assignment problem, p processors are requesting data from n different files. For simplicity, we assume that $n > p$. The assignment problem is to assign a processor for each file, such that when processors access their assigned files, the overall response time for the requests (total of I/O and communication times) is minimized. We estimate the I/O time by the amount of data accessed by a processor. The communication time is estimated by the amount of data that has to be transfered to/from other processors. So, the assignment problem is to find the optimal assignment of the processors to the files, such that $max_i\{\alpha \times I/O_i + \beta \times comm_i\}$ is minimized. α and β are constant values indicating the relative cost of I/O and communication in the system. I/O_i and $comm_i$ are the estimated I/O and communication times of processor i, respectively. They are estimated using the following formulas:

- $I/O_i = \sum_{j=1}^{n} a_{i,j}$ (the amount of data accessed by processor i)
- $comm_i = \sum_{j=1}^{n} |r_{i,j} - a_{i,j}|$ (the amount of data accessed by processor i subtracted from the amount of data requested by processor i).

In the above formulas, $r_{i,j}$ corresponds to the amount of data requested by processor i from file j. And, $a_{i,j}$ corresponds to the amount of data to be accessed by processor i from file j.

4.1.2 Complexity of the Assignment Problem

In this subsection, we show that the assignment problem as defined in the previous subsection is NP-complete. We prove that optimizing the I/O time (when $\alpha = 1$ and $\beta = 0$) is an NP-complete problem.

Claim: For arbitrary number of processors, number of files, and file sizes, finding the optimal assignment to minimize the I/O time is an NP-complete problem.

Sketch of the Proof: We prove the NP-completeness of the assignment problem using restriction. More specifically, we show that if we can solve the assignment problem in polynomial time, then we can solve the multiprocessor scheduling problem [11] in polynomial time, too. Multiprocessor scheduling problem is finding the m disjoint partitions of a finite task set A (A_1, A_2, ..., A_m), such that

$$max_i \quad \{\sum_{a \in A_i} length(a)\}$$

is minimized. Assume that we have a solver for the assignment problem that finds the optimal solution in polynomial time. Then, given a multiprocessor scheduling problem, we can easily transform the length of each task to a corresponding file size and use the assignment problem solver to solve the multiprocessor scheduling problem. Hence, we would be able to solve an arbitrary multiprocessor scheduling problem in polynomial time. Therefore, the assignment problem is NP-complete in the strong sense, because the multiprocessor scheduling problem is NP-complete in the strong sense for arbitrary m. □

4.2 Deciding the Number of Processors

Although the general assignment problem is NP-complete for arbitrary file sizes, for files of equal size, it can be

solved in polynomial time. In this section, we discuss how the assignments should be done in such a case. Specifically, we try to find the number of processors that will be used for accessing each file. In finding this value, we use the following basic model for the I/O time:

$$c_1 + c_2 * ((DataSize)/(Number\ of\ Processor))$$

where c_1 (corresponding to constant costs independent of the request size such as seek time) and c_2 (corresponding to transfer rate) are system dependent constant values. The intuition behind the model is that in a parallel environment the amount of data read by each processor will be reduced by increasing the number of processor. Hence if we omit the communication, there will be a decrease in the I/O time.

The total time for the request will be determined by the processor for which the I/O time is maximum (i.e., the one that completes the last). If we assume that the data sizes to be read from each file to be equal and that we assign k processors to each file, the response time will be equal to

$$m * c_1 + m * c_2 * ((DataSize)/(k)) \quad \text{Eq. 1}$$

where m is the number of files that the slowest processor is assigned to. If the assignments are done homogeneously among the processors, then

$$k = \frac{m*p}{n}$$

where p is the number of processors and n is the number of files. So, Equation 1 can be re-written as

$$m * c_1 + \frac{n}{p} * c_2 * DataSize$$

For a given number of processors and number of files, this expression takes its minimum for the minimum value of m. Since we have to assign at least one processor for each file the minimum value for m is $(n\ /\ p)$. That means, we have to assign only one processor to each file to minimize the response time. Note that, in the naive strategy, several processors are assigned to the files, which will increase the response time according to our model.

If the number of processors is larger than the number of files, then we have to assign one file to each processor only (the reverse of one processor for each file). The calculations are similar to the calculations given above. Note that in the calculations, we assume a homogeneous distribution of the files to the processors. If the file sizes are different, this might affect the response time significantly. This case is discussed in detail in Section 4.3.

Although we have found that we have to assign one processor to each file, we still have to determine which processor to assign to each file. Our selection of the processor will have a marginal effect on the I/O times, but it will have a significant effect on the communication times.

4.3 LP Model of the Assignment Problem

We are given a two-dimensional matrix $R = [r_{i,j}]$ such that an entry $r_{i,j}$ in R gives the amount of data requested by processor i from file j. In our ILP formulation, we would like to find the entries of a matrix $X = [x_{i,j}]$. An entry $x_{i,j}$ indicates whether the processor i is assigned to file j ($x_{i,j} = 1$) or not ($x_{i,j} = 0$). As mentioned earlier, although our selection of the processor for each file will not affect the I/O time, it will effect the communication time significantly. In the following discussion, n corresponds to the number of files to be accessed, p corresponds to the number of processors involved and we assume that $n > p$. So, we try to minimize

$\sum_{i=1}^{p} \sum_{j=1}^{n} r_{i,j} \times (1 - x_{i,j})$

This is because minimizing this expression will give us the total amount of data to be communicated. We will have the following constraints on $x_{i,j}$:

- $x_{i,j} \in \{0,1\}, \quad \forall i,j$

$x_{i,j}$ is a decision variable and should be either *assigned* or *not assigned*. Hence, the LP model of the problem is boolean or zero-one integer LP.

- $\sum_{i=1}^{p} x_{i,j} = 1, \quad \forall j$

In the previous section, we have found that each file will be assigned to exactly one processor. The above equation corresponds to this constraint. If the number of processors exceeds the number of files, this constraint becomes

- $\sum_{j=1}^{n} x_{i,j} = 1, \quad \forall i$

resulting in each processor being assigned to only one file.

- $\sum_{j=1}^{n} x_{i,j} = \frac{n}{p}, \quad \forall i$

This constraint makes sure that the assignments are homogeneous. It states that the number of files assigned to processor i equals to $\frac{n}{p}$. Here, we assume that regardless of the data size read from the files, we are going to assign same number of files to each processor. Although this assumption makes the calculation of the assignment variables easier, it might decrease the performance if the variance between the data sizes read is large. In such a case we can use the constraint

- $\sum_{j=1}^{n}(\sum_{k=1}^{p} r_{k,j}) \times x_{i,j} \leq P, \quad \forall i$

which means that the total data to be read by each processor should approximately be the same. In this expression, P corresponds to the average data size each processor should access. If the number of processors exceeds the number of files, then the constraint becomes

- $\sum_{i=1}^{p} x_{i,j} = F_j, \quad \forall j$

where F_j represents the number of processors to be assigned to file j. This number can easily be found using the amount of data read from the file j and all the other files. To calculate F_j, we use the following formula

$$F_k = p \times \frac{\sum_{i=1}^{p} r_{i,k}}{\sum_{i=1}^{p}\sum_{j=1}^{n} r_{i,j}}$$

p processors are distributed according to the portion of the file j with respect to all the data read.

Note that, in this model, the number of variables is equal to (*number of processors*) $\times$ (*number of files*). The above ZO-ILP model helps us understand the nature of the problem. But, we cannot use this model in a run-time library to make the processor-file assignments, since even the fastest LP-solvers will require extremely long running times to find a solution. Consequently, we developed two heuristics to solve the problem. We discuss these heuristics in the next section.

5 MCIO Heuristics

In this section, we explain two heuristics for making the processor-file assignments. The first one uses a sorting algorithm to find the assignment. The second one uses a maximal matching solver to make the assignments.

5.1 Greedy Heuristic

The first heuristic uses sorting for making the assignments. The main idea behind the heuristic is to assign the specific processors to the file, from which they are reading the largest amount of data. To achieve this, we first create an entry for every case, where processor i is reading a part of file j. Then, all these entries are sorted according to non-increasing values of the amount of data the processor i is reading from file j. As an example, the request list for the pattern shown in Figure 4 is given in Table 2. This list is formed using the request sizes given in Table 1.

The algorithm tries to pick the first element from the list and assign the processor to the corresponding file. If the file has already been assigned or if the processor has already completed the number of assignments it should have (i.e. the processor is full), then the entry is skipped and the next entry is checked. The algorithm continues until we make n assignments. If the entries in the list are finished before we make n assignments, we make the rest of the assignments randomly from the remaining processor pool. We do not have to pay attention in this step, because if the entries are finished, this means that all of the remaining processors are not reading any data from the remaining files. Thus, we can make the assignments arbitrarily because it will not change the amount of data communicated. The pseudo code of the algorithm is given in Figure 5. Returning to our example, the resultant assignments for the access in Figure 4 are given in Figure 6. Consequently, file 1 and file 4 are assigned to processor 2, file 2 and file 3 are assigned to processor 1, file 5 and file 8 are assigned to processor 3, and file 6 and file 7 are assigned to processor 4.

The insert_entry function in Figure 5 inserts a new entry into the list with the processor number i, file number j, and weight $r_{i,j}$. Similarly, remove_entry function deletes the entry from the list that is equal to its input parameter. After execution, the algorithm returns the list of processor–file pairs that have been assigned.

Table 1: The amount of data requested by each processor from each file for the access in Figure 4. The files are named row-major order starting from the upper left corner.

Processor Number	File Numbers							
	1	2	3	4	5	6	7	8
1	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
2	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
3	2.5 MB	3.2 MB	3.2 MB	1.1 MB	7.5 MB	9.6 MB	9.6 MB	3.3 MB
4	0 MB	0 MB	0 MB	0 MB	10 MB	12.8 MB	12.8 MB	4.4 MB

Table 2: The list formed for the example access in Figure 4.

Access List
$P1 \rightarrow F2, P1 \rightarrow F3, P2 \rightarrow F2, P2 \rightarrow F3, P4 \rightarrow F6, P4 \rightarrow F7, P1 \rightarrow F1, P2 \rightarrow F1, P4 \rightarrow F5, P3 \rightarrow F6, P3 \rightarrow F7,$ $P3 \rightarrow F5, P1 \rightarrow F4, P2 \rightarrow F4, P4 \rightarrow F8, P3 \rightarrow F8, P3 \rightarrow F2, P3 \rightarrow F3, P3 \rightarrow F1, P3 \rightarrow F4$

5.2 Maximal Matching Heuristic

The second heuristic uses a maximal matching solver. We have used the solver from the Netflow Solver Package [25]. The matching problem solver inside Netflow implements Gabow's N-cubed weighted matching algorithm [10]. This program is written by Ed Rothberg. To be able to use an existing maximal matching solver, we first need to build a graph representing the $r_{i,j}$s. Then, we need to modify the graph so that the solver gives the answer we seek.

The first step is straightforward. We build a graph G (V, E), where V contains a vertex for each processor and file. More specifically, if there are p processors and n files, then the graph will have $(n + p)$ vertices. The resulting vertices for the example access in Figure 4 are given in Figure 8(a). Then, we put an edge $(u, v) \in E$ with weight w, when the processor u reads w bytes of data from file v. The resulting graph is shown in Figure 8(b).

The existing matching problem solvers do not solve the exact problem we are interested in. They instead solve the maximum flow problem, given an input graph. For a bipartite graph G ($V_1 \bigcup V_2$, E) (note that, the graph given to the algorithm is bipartite), they try to find a different vertex in V_2 for each vertex in V_1 such that the sum of the weight of the edges between the selected node pairs is maximized (i.e., the flow is maximized). If the number of vertices in V_2 is larger then the number of vertices in V_1, then some of the vertices in V_2 will be left out. Similarly, if the number of vertices in V_1 is larger then the number of vertices in V_2, then some of the vertices in V_1 will be left out. Therefore, one has to make sure that the number of vertices for processors equals to the number of processors for files. We replicate the processor nodes to be able to make the number of processor vertices equal to the number of file vertices. For each file node, we replicate F_j times the processor nodes that have an edge to the node[1]. Since the solver assigns only one node for each file vertex, it gives the result we are looking for. This way, we guarantee that the solver makes n assignments. The resulting graph is given in Figure 8(c). Note that, in this example, we assume that F_j equals to two for every file node.

Once the graph has been constructed, we give it to the matching problem solver as input. The result of the solver is used as the assignments between the processor and the files. The assignments for the replicated nodes are interpreted as if they are assignments to the original node. The assignments made by the maximal matching heuristic for the access given in Figure 4 are given in Figure 7. Note that, if the number of processors is larger than the number of files, we replicate the file nodes instead of the processor nodes so that in the final result several processors might be assigned to the same file.

[1] The calculation of F_j is explained in Section 4.3.

```
GREEDY_ASSIGN (p, n, r_{i,j})
1.  /*p is the number of processors, n is
2.  the number of files, r_{i,j} represents
3.  the amount of data read by processor i
4.  from file j.  */
5.  begin
6.    for i=0 to p do
7.      for j=0 to n do
8.        if r_{i,j} ≠ 0 then
9.          insert_entry (i, j, r_{i,j})
10.       end if
11.     end for
12.   end for
13.   sort_entries_according_to (last_field)
14.   while (!list empty) AND
15.   (assignments_made < n) do
16.     entry = list_top
17.     if entry.processor full OR
18.       entry.file assigned then
19.       remove_entry (entry)
20.     else
21.       assign(entry.processor,entry.
22.       assignments_made ++
23.     end if
24.   end while
25.   assign the remaining files to
26.   remaining processors arbitrarily
27.   return all_assignments
28. end.
```

Figure 5: Greedy algorithm for making assignments algorithm sorts the requests from processors to file according to the access size. Then, it tries to assign the processors to the files such that the resulting assignment will result in a small communication overhead.

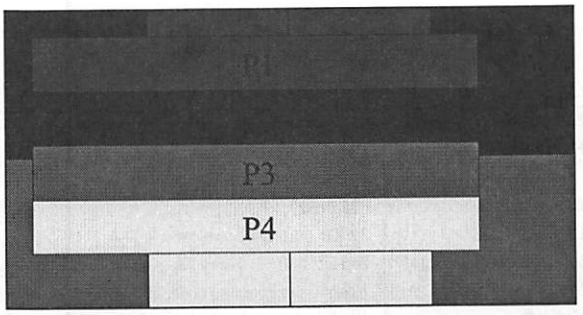


Figure 6: The result of the greedy heuristic. The colors denote the processor assigned.

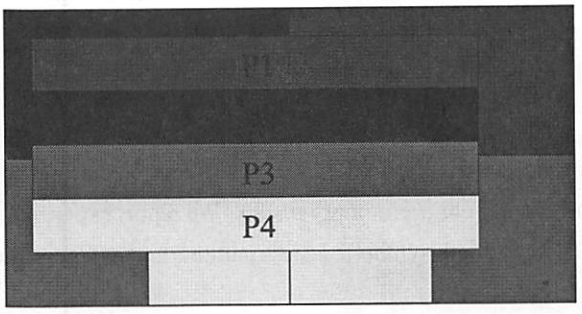


Figure 7: The result of the maximal matching heuristic. The colors denote the processor assigned.

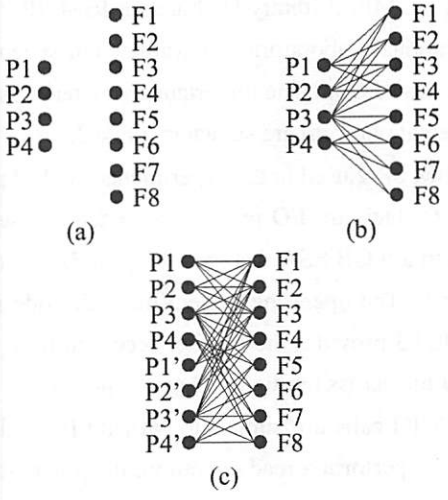


Figure 8: The creation of the input graph for the matching problem solver. The weights on the edges are left out for simplicity. (a) The initial graph with a node for each processor and a file. (b) The graph with an edge added for each case where processor i is reading data from file j. (c) The graph after the replication of the processor vertices to make the number of processor vertices and the file vertices to be equal.

Table 3: Platform used in the experiments.

Number of Processors	128 (120 compute nodes, 8 I/O nodes)
Processor Type	Compute Nodes: RS/6000 Model 370, I/O Nodes: RS/6000 Model 970
Clock Rate	332 MHz
L1 Cache	32 KB split, 2-way set-associative
L2 Cache	256 KB unified, 2-way set-associative
Memory Capacity	128 MB per compute node, 256 MB per I/O node
Network	100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI
Disk Space	9 GB per I/O node
Operating System	AIX 4.2.1
Parallel File System	PIOFS

6 Experiments

In this section, we discuss the experimental environment and discuss our preliminary results. We report experimental data for both synthetic access patterns and a large-scale scientific code.

6.1 Experimental Environment

We used the MPI-2 library [17] and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this section. The important characteristics of our experimental platform are shown in Table 3.

The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. Each I/O server is attached to a 9 GB SSA disk, resulting in 72 GB of total disk space. The operating system on each node is AIX 4.2.1. PIOFS provides the parallel access to files. It distributes a file across multiple I/O server nodes.

The MCIO calls are similar to MPI-IO [5] calls. For example, to perform a read operation, the processors call the

```
int MCIO_File_read_all (MPI_File *fh,
int filecount, void **buf, int *count,
MPI_Datatype *datatype, MPI_Status
*status)
```

routine. Note that the syntax of the call is very similar to `MPI_File_read_all` call in MPI-IO, except that MCIO routine takes an array of files (similarly, an array of buffers, an array of number of elements, and an array of data types) as argument. In addition, it takes an integer argument `filecount` indicating the number of files involved in the I/O request. Each array element corresponds to a request from a different file. Without the MCIO, calls to all the different files would have resulted in a separate MPI-IO call. In all experiments, we compare the performance of MCIO with that of traditional collective I/O.

6.2 Results for Synthetic Patterns

Figure 9 gives examples of the access patterns we experiment with. Two major types of experiments are conducted: row-major access and column-major access. In a row-major access, each processor accesses consecutive rows of the underlying data. In a column-major access, the array is distributed column-wise among the processors (i.e., each processor accesses a group of consecutive columns). For each category, we experiment with different number of processors and files.

For all the access patterns we experimented with, we evaluated the assignments resulting from the LP-model, greedy heuristic, and the maximal matching heuristic. The objective function and the constraints for the access in Figure 4 are given in Appendix A. The assignments for all the three methods were the same. This is mainly because of the constant file size we have used in the experiments. Due to this invariance of the results, we do not present separate execution times for these methods, but discuss the advantages and disadvantages of each. Note that, the assignments of greedy algorithm and the maximal matching heuristic will differ only in their communication time, not the I/O time. Therefore, there is some difference between the response times of the assignments made by the two heuristic. The greedy algorithm is faster to find the assignments than the maximal matching algorithm. Hence, in cases where the access sizes from different processors to different files are the same (similar to the access patterns in Figure 9), we recommend the use of the greedy algorithm. On the other hand, if the request sizes from different files have a significant difference, the maximal matching algorithm always gives a better communication time. Also, if the total amount of data requested by differ-

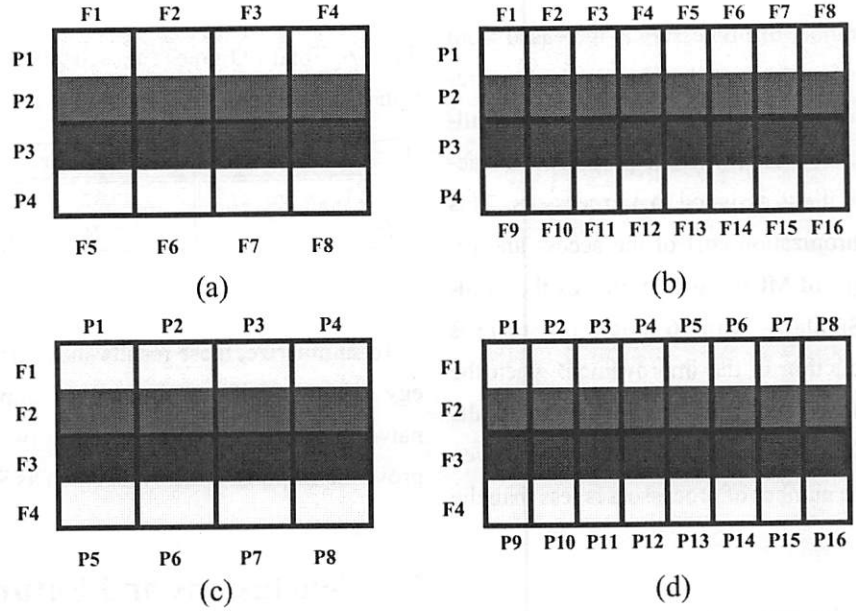


Figure 9: Examples of the experimented access patterns: (a) Four processor accessing data row-wise from 8 files, (b) Four processor accessing data row-wise from 16 files, (c) 8 processor accessing data column-wise from four files, (d) 16 processor accessing data column-wise from four files.

Table 4: The improvements for row major accesses (Figure 9(a) and (b)) in [%] with respect to the naive I/O access.

Number of Files	Number of Processors		
	4	8	16
4	49.97	51.22	49.97
8	52.28	78.33	77.41
16	53.42	80.12	86.67
32	54.33	80.52	87.18

Table 5: The improvements for column major accesses (Figure 9(c) and (d)) in [%] with respect to the naive I/O access.

Number of Files	Number of Processors		
	4	8	16
4	41.26	55.76	49.04
8	43.90	76.49	79.26
16	45.21	77.59	80.28
32	45.84	78.07	80.82

ent processors have a large variance, maximal matching algorithm gives better performance. Hence, in such cases maximal matching algorithm should be employed.

In the first set of experiments conducted, the file size is set to 32 MB, representing a two-dimensional matrix of 1024×2048 floating points for the case of four processors reading data from four files. Therefore, the total amount of data read in this case equals to 128 MB. When the number of files is increased, the total amount of data accessed increases linearly.

6.2.1 Row-Major Accesses

The results for the experiments with row-wise access patterns are summarized in Table 4. The table gives the improvements achieved by MCIO technique compared to a naive access using the CIO technique for each accessed file. The MCIO is able to improve the response time in the base case (4 processors reading data from 4 files) by 49.97% over a naive access pattern, which performs CIO for each of the 4 files separately. The results also reveal that as the number of processors increases, the improvement also increases. In addition, when the number of files is increased, the improvement increases. An exception

occurs when the number of processors is increased from 8 to 16 when 4 files are accessed. The reason for a reduction in the improvement is the small parallelism available for this access. Note that, when 16 processors are accessing 4 files, each file is assigned to 4 processors. This increases the synchronization cost of the access and reduces the advantages of MCIO, which utilizes the available parallelism. Similarly, with 16 processors, there is an insignificant reduction in the improvement when the number of files is increased from 8 to 16. These results indicate that MCIO brings significant improvement even in the case where the number of processors is less than the number of files.

6.2.2 Column-Major Accesses

Table 5 summarizes the experimental results for column-wise access patterns. Although, the performance improvement by MCIO is slightly less for column-major accesses, MCIO still brings substantial amount of improvement of the naive CIO technique. Specifically, MCIO is able to improve the CIO performance by as much as 80.82% when 16 processors are requesting data from 32 files. Similar to row-major accesses, MCIO brings better improvement over the naive CIO performance when the number of processors or files are increased. The case where the number of processors is increased from 8 to 16 for reading 4 files is again an exception to the general trend.

6.3 Results for a Scientific Application

We have also applied the MCIO technique to improve the I/O performance of the *astro3d* [15] application. Table 6 shows the results for this three-dimensional astrophysics application. *Astro3d* accesses six different variables from a single file. In the original application, this corresponds to six different collective I/O calls. Using MCIO, same data can be accessed using a single call, thereby the parallelism in the system is better utilized. For 4 processors, this results in a reduction of the I/O time by 38.52%. For 8 processors, the improvement of MCIO over traditional collective I/O increases to 62.96%.

Table 6: Total I/O times (in seconds) for *astro3d* application (Data set size is 8 MB).

	4 processors	8 processors
Collective I/O	3.33	3.51
MCIO	2.04	1.30

To summarize, these results show that the MCIO strategy brings significant amount of improvement over a naive CIO access method. Specifically, we are able to improve the response time by as much as 87.18%.

7 Conclusions and Future Work

In this paper, we have introduced an I/O optimization technique called multi-collective I/O. As the gap between the performance of the processor and the storage subsystem increases, more aggressive optimizations are needed to be able to feed the processor with enough data. Several scientific applications exhibit poor storage access patterns, and hence optimizations like MCIO can bring significant improvement in the execution time of such applications.

We have first shown that finding the optimal access pattern in MCIO is an NP-complete problem. Then, we have presented two heuristics to perform this task: a greedy algorithm that uses sorting and a graph algorithm that uses a matching problem solver. Then, using synthetic benchmarks and a scientific application, we have shown that MCIO can bring substantial amount of improvement in the I/O response time over a collective I/O technique. Specifically, MCIO was able to improve the response time by as much as 87.18%.

Our current work focuses on using the scheduling obtained through our approach in developing powerful I/O prefetching techniques. Also in our agenda is evaluating the effectiveness of MCIO in a sub-file based environment. We also plan to design and implement an optimizing compiler framework for generating I/O-optimized code automatically using the MCIO interface provided by our runtime library. Such a compiler will relieve applica-

tion developers from low-level details of file systems and runtime libraries, and let them focus instead on high-level (application-specific) aspects of their codes.

References

[1] Bagrodia, R., Chien, A., Hsu, Y., Reed, D. Input/output: Instrumentation, Characterization, Modeling and Management Policy. *Tech. Rep. CCSF-41*, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[2] Baru, S. *Storage Resource Broker (SRB) Reference Manual*. Enabling Technologies Group, San Diego Supercomputer Center, La Jolla, CA, August 1997.

[3] Beynon, M., Kurc, T., Sussman, A., Saltz, J. Design of a Framework for Data-Intensive Wide-Area Applications. In *Proc. of the 9th Heterogeneous Computing Workshop (HCW2000)*, Cancun, Mexico, May 2000.

[4] Choudhary, A., Bordawekar, R., Harry, M., Krishnaiyer, R., Ponnusamy, R., Singh, T., and Thakur, R. PASSION: parallel and scalable software for input-output. *NPAC Technical Report SCCS-636*, Sept 1994.

[5] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J., Snir, M., Traversat, B., and Wong, P. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Parallel and Distributed Systems*, IPPS'95, Santa Barbara, CA, April 1995.

[6] Corbett, P., Feitelson, D., Prost, J-P., Almasi, G., Baylor, S. J., Bolmarcich, A., Hsu, Y., Satran, J., Snir, M., Colao, R., Herr, B., Kavaky, J., Morgan, T., and Zlotek, A. Parallel file systems for the IBM SP computers, *IBM Systems Journal*, Vol. 34, No. 2, pp. 222–248, Jan 1995.

[7] Coyne, R. A., Hulen, H., and Watson R.. The high performance storage system. In *Proc. Supercomputing 93*, Portland, OR, November 1993.

[8] Crandall, P., Aydt, R., Chien, A., Reed, D. Input/output characteristics of scalable parallel applications. In *Proc. of Supercomputing '95*, Dec. 1995.

[9] Cypher, R., Ho, A., Konstantinidou, S., Messina, P. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proc. of the 20th International Symposium on Computer Architecture*, 1993, pp. 2-13.

[10] Gabow, H. Implementation of Algorithms for Maximum Matching on Non-bipartite Graphs. *Ph.D Thesis*, Stanford University, 1973.

[11] Garey, M. R. and Johnson, D. S. Computers and Intractability, *A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[12] Karpovich, J. F., Grimshaw, A. S., and French, J. C. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proc. the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 191–204, Oct 1994.

[13] Kotz, D. Disk-directed I/O for MIMD multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth *TR PCS-TR94-226* on November 8, 1994.

[14] Kotz, D. Multiprocessor file system interfaces. In *Proc. of the Second International Conference on Parallel and Distributed Information Systems*, pp. 194-201, 1993.

[15] Malagoli, A. ASTRO3D - 2.0. The ASCI Flash Center at Chicago, IL, Sep. 1998.

[16] Memik, G., Kandemir, M. T., Choudhary, A. APRIL: A Run-Time Library for Tape Resident Data. In *Proc. of 8.NASA Goddard Conference on Mass Storage Systems and Technologies*, Baltimore, MD, April 2000.

[17] Message Passing Interface Forum. MPI-2: Extension to the message passing interface. *Technical report*, University of Tennessee, July 1997. URL: http://www.mpi-forum.org.

[18] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C., Best, M. File-Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems*, October 1996 (Vol. 7, No. 10), pp. 1075-1089.

[19] Nitzberg, B. J. Collective Parallel I/O. *PhD thesis*, Department of Computer and Information Science, University of Oregon, December 1995.

[20] Nitzberg, B. Performance of the iPSC/860 Concurrent File System. *Technical Report RND-92-020*, NAS Systems Division, NASA Ames, Dec. 1992.

[21] Rosario, J., Bordawekar, R., Choudhary, A. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. the 1993 IPPS Workshop on Input/Output in Paral. Comp. Sys.*, April 1993.

[22] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., Winslett, M. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing'95,* December 1995

[23] Seamons, K. E., Winslett, M. Efficient sub-image extraction using Panda. URL: http://drl.cs.uiuc.edu/panda/demo/

[24] Shen, X., Liao, W., Choudhary, A., Memik, G., Kandemir, M., More, S., Thiruvathukal, G., and Singh, A. A Novel Application Development Environment for Large-Scale Scientific Computations. In *Proc. of International Conference on Supercomputing*, Santa Fe, New Mexico, May, 2000.

[25] URL: ftp://dimacs.rutgers.edu/pub/netflow/

[26] Thakur, R., Lusk, E., Gropp, W. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. *Technical Memorandum ANL/MCS-TM-234*, Mathematics and Computer Science Division, Argonne National Laboratory, IL, Revised July 1998.

APPENDIX A *An Example LP Model*

Ee give the LP model for the access in Figure 4 and also present the result of the solver. We have used the CPLEX as the LP-solver, hence the model and the results are given using the CPLEX format. The LP-model is as follows:

```
minimize
M
subject to
c1:  x1_1 + x2_1 + x3_1 + x4_1 = 1
c2:  x1_2 + x2_2 + x3_2 + x4_2 = 1
c3:  x1_3 + x2_3 + x3_3 + x4_3 = 1
c4:  x1_4 + x2_4 + x3_4 + x4_4 = 1
c5:  x1_5 + x2_5 + x3_5 + x4_5 = 1
c6:  x1_6 + x2_6 + x3_6 + x4_6 = 1
c7:  x1_7 + x2_7 + x3_7 + x4_7 = 1
c8:  x1_8 + x2_8 + x3_8 + x4_8 = 1
c9:  x1_1 + x1_2 + x1_3 + x1_4 + x1_5 +
x1_6 + x1_7 +x1_8 = 2
c10:  x2_1 + x2_2 + x2_3 + x2_4 + x2_5 +
x2_6 + x2_7 + x2_8 = 2
c11:  x3_1 + x3_2 + x3_3 + x3_4 + x3_5 +
x3_6 + x3_7 + x3_8 = 2
c12:  x4_1 + x4_2 + x4_3 + x4_4 + x4_5 +
x4_6 + x4_7 + x4_8 = 2
c13:  M + 10x1_1 + 12x1_2 + 12x1_3 +
5x1_4 + 10x2_1 + 12x2_2 + 12x2_3 + 5x2_4
+ 2x3_1 + 3x3_2 + 3x3_3 + 2x3_4 + 7x3_5
+ 9x3_6 + 9x3_7 + 3x3_8 + 10x4_5 +
12x4_6 + 12x4_7 + 5x4_8 = 155
binary
x1_1 x1_2 x1_3 x1_4 x1_5 x1_6 x1_7 x1_8
x2_1 x2_2 x2_3 x2_4 x2_5 x2_6 x2_7 x2_8
x3_1 x3_2 x3_3 x3_4 x3_5 x3_6 x3_7 x3_8
x4_1 x4_2 x4_3 x4_4 x4_5 x4_6 x4_7 x1_8
end
```

Note that, we have rounded the request sizes (`c13`) from Table 1. In the above model, $x_{i,j}$'s are denoted by xi_j and M is the objective function. The first 8 constraints correspond to the $\sum_{i=1}^{p} x_{i,j} = 1$ constraint in Section 4.3. Constraints 9 through 12 correspond to the $\sum_{j=1}^{n} x_{i,j} = \frac{n}{p}$ constraint which guarantees homogeneous distribution of the files among processors.

The result for the above model is as follows:

Variable Name	Solution Value
M	82.000000
x2_1	1.000000
x2_2	1.000000
x1_3	1.000000
x1_4	1.000000
x3_5	1.000000
x4_6	1.000000
x4_7	1.000000
x3_8	1.000000

All other variables in the range 1-33 are zero.

The value of 82 for `M` is the optimal communication that can be achieved. The variables $x_{2,1}$, $x_{2,2}$, $x_{1,3}$, $x_{1,4}$, $x_{3,5}$, $x_{4,6}$, $x_{4,7}$, and $x_{3,8}$ are 1. The remaining variables are zero. Note that, this assignment is the same as the maximal matching heuristic has made, which is shown in Figure 7.

Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics

Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, Gregory R. Ganger
Carnegie Mellon University

Abstract

Track-aligned extents (*traxtents*) utilize disk-specific knowledge to match access patterns to the strengths of modern disks. By allocating and accessing related data on disk track boundaries, a system can avoid most rotational latency and track crossing overheads. Avoiding these overheads can increase disk access efficiency by up to 50% for mid-sized requests (100–500 KB). This paper describes traxtents, algorithms for detecting track boundaries, and some uses of traxtents in file systems and video servers. For large-file workloads, a version of FreeBSD's FFS implementation that exploits traxtents reduces application run times by up to 20% compared to the original version. A video server using traxtent-based requests can support 56% more concurrent streams at the same startup latency and buffer space. For LFS, 44% lower overall write cost for track-sized segments can be achieved.

1 Introduction

Rotating media has come full circle, so to speak. The first uses of disks in the 1950s ignored the effects of geometry in the interest of achieving a working system. Later, algorithms were developed that paid attention to disk geometry in order to improve disk efficiency. These algorithms were often hard-coded and hardware-specific, making them fragile across generations of hardware. To address this, a layer of abstraction was standardized between operating systems and disks, virtualizing disk storage as a flat array of fixed-sized blocks. Unfortunately, this abstraction hides too much information, making the OS's task of maximizing disk efficiency more difficult than necessary.

File systems and databases attempt to mitigate the ever-present disk performance problem by aggressively clustering on-disk data and by issuing fewer, larger disk requests. This is usually done with only a vague understanding of disk characteristics, focusing on the notion that bigger requests are better because they amortize per-request positioning delays over larger data transfers. Although this notion is generally correct, there are performance and complexity costs associated with making requests larger and larger. For video servers, ever-larger requests increase both buffer space requirements and stream initiation latency [6, 7, 22]. Log-structured file systems (LFS) incur higher cleaning overheads as segment size increases [5, 24, 33]. Even for general file system operation, allocation of very large sequential regions competes with space management robustness [25], and very large accesses may put deep prefetching ahead of foreground requests. Also, large requests can be used for small files by grouping their contents [14, 15, 17, 32, 33], but larger requests require grouping more files with weaker inter-relationships. These examples all indicate that achieving higher disk efficiency with smaller request sizes would be valuable.

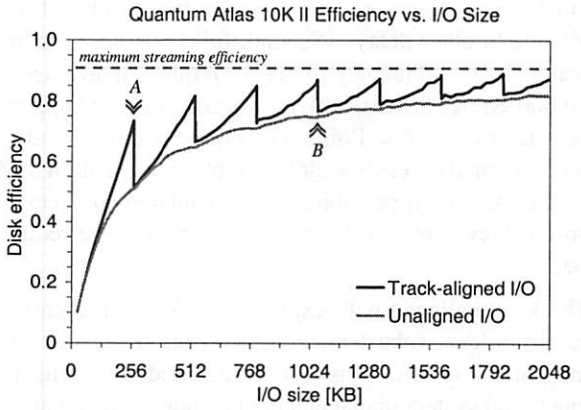


Figure 1: **Measured advantage of track-aligned access over unaligned access.** *Disk efficiency* is the fraction of total access time spent moving data to or from the media. The maximum streaming efficiency is less than 1.0, because no data is transferred when switching from one track to the next. The track-aligned and unaligned lines show disk efficiency for random, constant-sized reads within a Quantum Atlas 10K II's first zone (264 KB per track). Point A highlights the higher efficiency of track-aligned access (0.73, or 82% of the maximum) over unaligned access for a track-sized request. Point B shows where unaligned I/O efficiency catches up to the track-aligned efficiency at Point A. The peaks in the track-aligned curve correspond to multiples of the track size.

This paper describes and analyzes track-aligned extents (*traxtents*), extents that are aligned and sized so as to

match the corresponding disk track size. By exploiting a small amount of disk-specific knowledge in this way, a system can significantly increase the efficiency of mid-to-large requests (100 KB and up). Traxtent-aware access yields up to 50% higher disk efficiency, quantified as the the fraction of total access time spent moving data to or from the media.

The efficiency improvement stems from two main sources. First, track-aligned access minimizes the number of track switches, whose times have not decreased much over the years and are now significant (0.6–1.1 ms) relative to other delays. Second, full-track access eliminates rotational latency (3 ms per request on average at 10,000 RPM) for disk drives whose firmware supports zero-latency access. Point A of Figure 1 shows random track-aligned accesses yielding an efficiency within 82% of the maximum possible, whereas unaligned accesses only achieve 56% of the best-case for the same request size.

The key challenge with exploiting disk-specific knowledge is clean, robust integration: complexity must be minimized, systems must not become tied to specific devices, and system management must not be made harder. These concerns can be addressed by minimizing the disk-specific details needed, determining them automatically for any disk, and incorporating them in a generic fashion. This paper promotes the use of track boundaries, describes algorithms for detecting them automatically, and describes how they can be cleanly integrated into existing systems. In particular, simply changing a file system to support variable-sized extents is sufficient — the file system code need not depend on any particular disk's track boundaries. Further, variable-sized extents allow a file system to accomodate other boundary-related goals, such as matching writes to stripe boundaries in order to avoid RAID 5 read-modify-write operations [9].

This paper extensively explores track-based access. Detailed disk measurements show increased disk efficiency and reduced access time variance. They also identify system requirements that must be satisfied to achieve the highest efficiency. A prototype implementation of a traxtent-aware FFS file system in FreeBSD 4.0 illustrates the minimal changes needed and the resulting benefits. For example, when accessing two large files concurrently, the traxtent-aware FFS yields 20% higher performance compared to current defaults. For streaming media workloads, a video server can support either 56% more concurrent streams at the same startup latency or a 5× reduction in startup latency and buffer space at the maximum number of concurrent streams. Finally, we compute 44% lower overall write cost for track-sized segments in LFS.

Although track boundary knowledge was used for allocation and access decisions in some pre-SCSI systems, no current or recent system that we are aware of does so. This paper makes several enabling contributions:

1. It identifies and quantifies the benefits of track-based access on modern disks, showing up to 50% increases in efficiency. This is a compelling performance boost.

2. It introduces new algorithms for automatically detecting track boundaries. This task is more difficult than might be expected, because of zoned recording and media defect management.

3. It describes a minimal set of changes needed to use track boundary knowledge in an existing file system. This experience supports our contention that exploiting disk-specific knowledge appropriately need not introduce hardware dependences.

4. It shows that the disk efficiency benefits translate into application performance increases in some real cases, including streaming media services and large file manipulations.

The remainder of this paper is organized as follows. Section 2 motivates track-based access by describing the technology trends and expected benefits in more detail. Section 3 describes system changes required for traxtents. Section 4 describes our implementation of traxtents in FreeBSD. Section 5 evaluates traxtents under a variety of circumstances. Section 6 discusses related work. Section 7 summarizes this paper's contributions.

2 Track-based Disk Access

In determining what data to read and write when, system software attempts to maximize overall performance in the face of two competing pressures. On the one hand, the underlying disk technology pushes for larger request sizes in order to maximize disk efficiency. Specifically, time-consuming mechanical delays can be amortized by transferring large amounts of data between each repositioning of the disk head. For example, Point B of Figure 1 shows that reading or writing 1 MB at a time results in a 75% disk efficiency for normal (track-unaligned) access. On the other hand, resource limitations and imperfect information about future accesses impose costs on the use of very large requests.

This section discusses the system-level issues that push for smaller request sizes, the disk characteristics that make track-based accesses particularly efficient, and the types of applications that will benefit most from track-based disk access.

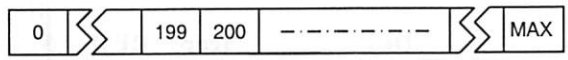
2.1 Limitations on request size

Four system-level factors oppose the use of ever-larger requests: (1) responsiveness, (2) limited buffer space, (3) irregular access patterns, and (4) storage space management.

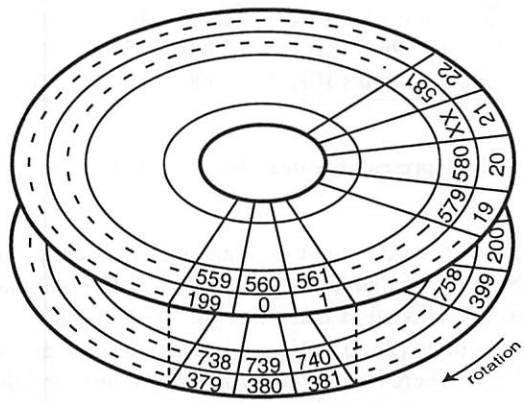
Responsiveness. Although larger requests increase disk efficiency, they do so at the expense of higher latency. This trade-off between efficiency and responsiveness is a recurring theme in computer systems, and it is particularly steep for disk systems. The latency increase can manifest itself in several ways. At the local level, the non-preemptive nature of disk requests combined with the long access times of large requests (35–50 ms for 1 MB requests) can result in substantial I/O wait times for small, synchronous requests. This problem has been noted for both FFS and LFS [5, 37]. At the global level, grouping substantial quantities of data into large disk writes usually requires heavy use of write-back caching. Although application performance is usually decoupled from the eventual write-back, application changes are not persistent until the disk writes complete. Making matters worse, the amount of data that must be delayed and buffered to achieve large enough writes continues to grow. As another example, many video servers fetch video segments in carefully-scheduled rounds of disk requests. Using larger disk requests increases the time for each round, which increases the time required to start streaming a new video. Section 5.4 quantifies the start-up latency required for modern disks.

Buffer space. Although memory sizes continue to grow, they remain finite. Larger disk requests stress memory resources in two ways. For reads, larger disk requests are usually created by fetching more data farther in advance of the actual need for it; this prefetched data must be buffered until it is needed. For writes, larger disk requests are usually created by holding more data in a write-back cache until enough contiguous data is dirty; this dirty data must be buffered until it is written to disk. The persistence problem discussed above can be addressed with non-volatile RAM, but the buffer space issue will remain.

Irregular access patterns. Large disk requests are most easily generated when applications use regular access patterns and large files. Although sequential full-file access is relatively common [1, 29, 45], most data objects are much smaller than the disk request sizes needed to achieve good disk efficiency. For example, most files are well below 32 KB in size in UNIX-like systems [15, 40] and below 64 KB in Microsoft Windows systems [12, 45]. Directories and file attribute structures are almost always much smaller. To achieve sufficiently large disk requests in such environments, access patterns



(a) **System's view of storage.**



(b) **Mapping of LBNs onto physical sectors.**

Figure 2: **Standard system view of disk storage and its mapping onto physical disk sectors.** (a) illustrates the linear sequence of logical blocks, often 512 bytes, that the standard disk protocols expose. (b) shows one example mapping of those logical block numbers (LBNs) onto the disk media. The depicted disk drive has 200 sectors per track, two media surfaces, and track skew of 20 sectors. Logical blocks are assigned to the outer track of the first surface, the outer track of the second surface, the second track of the first surface, and so on. The track skew accounts for the head switch delay to maximize streaming bandwidth. The picture also shows a defect between the sectors with LBNs 580 and 581, depicted as XX, which has been handled by slipping. Therefore, the first LBN on the following track is 599 instead of 600.

across data objects must be predicted at on-disk layout time. Although approaches to grouping small data objects have been explored [14, 15, 17, 32, 33], all are based on imperfect heuristics, and thus they rarely group things perfectly. Even though disk efficiency is higher, misgrouped data objects result in wasted disk bandwidth and buffer memory, since some fetched objects will go unused. As the target request size grows, identifying sufficiently strong inter-relationships becomes more difficult.

Storage space management. Large disk requests are only possible when closely related data is collocated on the disk. Achieving this collocation requires that on-disk placement algorithms be able to find large regions of free space when needed. Also, when grouping multiple data objects, growth of individual data objects must be accommodated. All of these needs must be met with little or no information about future storage allocation and deallocation operations. Collectively,

Disk	Year	RPM	Head Switch	Avg. Seek	512B Sectors per Track	Number of Tracks	Capacity
HP C2247	1992	5400	1 ms	10 ms	96–56	25649	1 GB
Quantum Viking	1997	7200	1 ms	8.0 ms	216–126	49152	4.5 GB
IBM Ultrastar 18 ES	1998	7200	1.1 ms	7.6 ms	390–247	57090	9 GB
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195	116340	18 GB
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224	60126	9 GB
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286	103750	18 GB
Quantum Atlas 10K II	2000	10000	0.6 ms	4.7 ms	528–353	52014	9 GB

Table 1: **Representative disk characteristics.** Note the small change in head switch time relative to other characteristics.

these facts create a complex storage management problem. Systems can address this problem with combinations of pre-allocation heuristics [4, 18], on-line reallocation actions [23, 33, 41], and idle-time reorganization [2, 24]. There is no straightforward solution and the difficulty grows with the target disk request size, because more related data must be clustered.

2.2 Disk characteristics

Modern storage protocols, such as SCSI and IDE/ATA, expose storage capacity as a linear array of fixed-sized blocks (Figure 2(a)). By building atop this abstraction, OS software need not concern itself with complex device-specific details, and code can be reused across the large set of storage devices that use these interfaces (e.g., disk drives and disk arrays). Likewise, by exposing only this abstract interface, storage device vendors are free to modify and enhance their internal implementations. Behind this interface, the storage device must translate the logical block numbers (LBNs) to physical storage locations. Figure 2(b) illustrates this translation for a disk drive, wherein LBNs are assigned sequentially on each track before moving to the next. Disk drive advances over the past decade have conspired to make the track a sweet-spot for disk efficiency, yielding the 50% increase at Point A of Figure 1.

Head switch. A head switch occurs when a single request accesses a sequence of LBNs whose on-disk locations span two tracks. This head switch consists of turning on the electronics for the appropriate read/write head and adjusting its position to account for inter-surface alignment imperfections. The latter step requires the disk to read servo information to determine the head's location and then to shift the head towards the center of the second track. In the example of Figure 2(b), head switches occur between LBNs 199 and 200, 399 and 400, and 598 and 599.

Even compared to other disk characteristics, head switch time has improved little in the past decade. While disk rotation speeds have improved by 3× and average seek times by 2.5×, head switch times have decreased by only 20–40% (see Table 1). At 0.6–1.1 ms, a head switch now takes about 1/5 of a revolution for a 15,000 RPM disk. This trend has increased the significance of head switches. Further, this trend is expected to continue, because rapid decreases in inter-track spacing require increasingly precise head positioning.

Naturally, not all requests span track boundaries. The probability of a head switch, P_{hs}, depends on workload and disk characteristics. For a request of N sectors and a track size of SPT sectors, $P_{hs} = (N-1)/SPT$, assuming that the requested locations are uncorrelated with track boundaries. For example, with 64 KB requests ($N = 128$) and an average track size of 192 KB ($SPT = 384$), a head switch occurs for every third access, on average. With N approaching SPT, almost every request will involve a head switch, which is why we refer to conventional systems as "track-unaligned" even though they are only "track-unaware". In this situation, track-aligned access improves the response time of most requests by the 0.6–1.1 ms head switch time.

Zero-latency access. A second disk feature that pushes for track-based access is zero-latency access, also known as immediate access or access-on-arrival. When disk firmware wants to read N contiguous sectors, the simplest approach is to position the head (by a combination of seek and rotational latency) to the first sector and read the N sectors in ascending LBN order. With zero-latency access support, disk firmware can read the N sectors from the media into its buffers in any order. In the best case of reading exactly one track, the head can start reading data as soon as the seek is completed; no rotational latency is involved because all sectors on the track are needed. The N sectors are read into an intermediate buffer, assembled in ascending LBN order, and sent to the host. The same concept applies to writes, except that data must be moved from host memory to the disk's buffers before it can be written onto the media.

As an example of zero-latency access on the disk from Figure 2(b), consider a read request for LBNs 200–399.

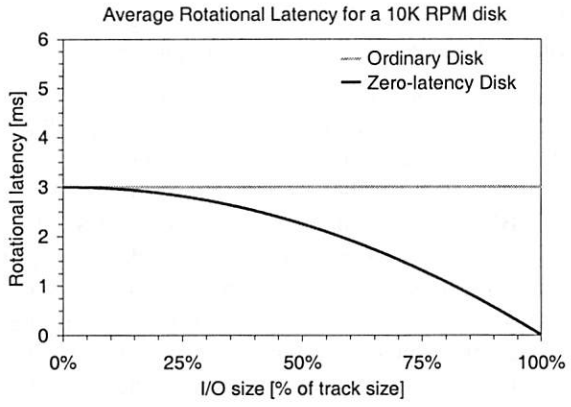


Figure 3: **Average rotational latency for ordinary and zero-latency disks as a function of track-aligned request size.** The request size is expressed as a percentage of the track size.

First, the head is moved to the track containing these blocks. Suppose that, after the seek, the disk head is positioned above the sector containing LBN 380. A zero-latency disk can immediately read LBNs 380–399. It then reads the sectors with LBNs 200–379. In this way, the entire track can be read in only one rotation even though the head arrived in the "middle" of the track.

The expected rotational latency for a zero-latency disk decreases as the request size increases, as shown in Figure 3. Therefore, a request to the zero-latency access disk for all SPT sectors on a track requires only one revolution after the seek. An ordinary disk, on the other hand, has an expected rotational latency of $(SPT - 1)/(2 \cdot SPT)$, or approximately 1/2 revolution, regardless of the request size and thus a request requires anywhere from one to two (average of 1.5) revolutions.

2.3 Putting it all together

For requests around the track size (100–500 KB), the potential benefit of track-based access is substantial. A track-unaligned access for SPT sectors involves four delays: seek, rotational latency, SPT sectors worth of media transfer, and head switch. An SPT-sector track-aligned access eliminates the rotational latency and head switch delays. This reduces access times for modern disks by 3–4 ms out of 9–12 ms, resulting in a 50% increase in efficiency.

Of course, the real benefit provided by track-based access depends on the workload. For example, a workload of random small requests, as characterizes transaction processing, will see minimal improvement because request sizes are too small. At the other end of the spectrum, a system that sequentially reads a single large file will also see little benefit, because positioning costs can be amortized over megabyte sized transfers and the disk's prefetching logic will ensure that this occurs. Track-based access provides the highest benefit to applications with medium-sized I/Os. One set of examples is streaming media services, such as video servers, MP3 servers, and CDN caches. Another includes storage components (e.g., Network Appliance's filers [19], HP's AutoRAID [47], or EMC's Symmetrix) that map data to disk locations in mid-sized chunks. Section 5 explores several concrete examples of such applications.

3 Traxtent-aware System Design

Track-based disk access is a design option for any system component that allocates disk locations and generates disk requests. In some systems, like the one used in our experiments, these decisions are made in the system software (e.g., file system) of a workstation, file server, or content-caching appliance. In others, the system software decisions are overridden by a logical disk [11] or a high-end disk array controller [42, 47], using some sort of mapping table to translate requested LBNs to internal disk locations. Track-based disk access is appropriate within any of these systems, and it requires relatively minor changes to existing systems. This section discusses practical design considerations involved with these changes.

3.1 Extracting track boundaries

In order to use track boundary information, a system must first obtain it. Specifically, a system must know the range of LBNs that map onto each track. Under ideal circumstances, the disk would provide this information directly. However, since current SCSI and IDE/ATA disks do not, the track boundaries must be determined experimentally.

Extracting track boundaries is made difficult by the internal space management algorithms employed by disk firmware. In particular, three aspects complicate the basic LBN-to-physical mapping pattern. First, because outer tracks have greater circumference than inner tracks, modern disks record more sectors on the outer tracks. Typically, the set of tracks is partitioned into 8–20 subsets (referred to as zones or bands), each with a different number of sectors per track. Second, because some amount of defective media is expected, some fraction of the disk's sectors are set aside as spare space for defect management. This spare space disrupts the pattern even when there are no defects. Worse, there are a wide array of spare space schemes (e.g., spare sectors per track, spare sectors per cylinder, spare tracks per zone, spare space at the end of the disk, etc.); we have observed over 10 distinct schemes in different disk makes and models. Third, when defects exist, the default LBN-to-physical mapping is modified to avoid the defective re-

gions. Defect avoidance is handled in one of two ways: *slipping*, wherein the LBN-to-physical mapping is modified to simply skip the defective sector, and *remapping*, wherein the LBN that would be located in a defective sector is instead located in a spare sector. Slipping is more efficient and more common, but it affects the mappings of subsequent LBNs.

Although track detection can be difficult, it need be performed only once. Track boundaries only change in use if new defects "grow" on the disk, which is rare after the first 48 hours of operation [30].

3.2 Allocation and access

To utilize track boundary information, the algorithms for on-disk placement and request generation must support variable-sized extents. Extent-based file systems, such as NTFS [28] and XFS [43], allocate disk space to files by specifying ranges of LBNs (extents) associated with each file. Such systems lend themselves naturally to track-based alignment of data: during allocation, extent ranges can be chosen to fit track boundaries. Block-based file systems, such as Ext2 [4] and FFS [25], group LBNs into fixed-size allocation units (blocks), typically 4 KB or 8 KB in size.

Block-based systems can approximate track-sized extents by placing sequential runs of blocks such that they never span track boundaries. This approach wastes some space when track sizes are not evenly divisible by the block size. However, this space is usually less than 5% of total storage space and could be reclaimed by the system for storing inodes, superblocks, or fragmented blocks. Alternately, this space can be reclaimed if the cache manager can be modified to handle partially-valid and partially-dirty blocks.

Like any clustering storage system, a traxtent-based system must address aging and fragmentation and the standard techniques apply: pre-allocation [4, 18], on-line reallocation [23, 33, 41], and off-line reorganization [2, 24]. For example, when a system determines that a large file is being written, it may be useful to reserve (preallocate) entire traxtents even when writing less than a traxtent worth of data. The same holds when grouping small files [15, 32]. When the file system becomes aged and fragmented, on-line or off-line reorganization can be used to re-optimize the on-disk layout. Such reorganization can also be used for retrofitting pre-existing disk partitions or adapting to a replacement disk. The point of this paper is that traxtents are a good target layout for these techniques.

After allocation routines are modified to situate data on track boundaries, system software must also be extended to generate traxtent requests whenever possible. Usually, this will involve extending or clipping prefetch and write-back requests based on track boundaries.

Our experimentation uncovered an additional design consideration: current systems only realize the full benefit of track-based requests when using command queueing at the disk. Although zero-latency disks can access LBNs on the media in any order, current SCSI and IDE/ATA controllers only allow for in-order delivery to or from the host. As a result, bus transfer overheads hide some of the benefit of zero-latency access. By having multiple requests outstanding at the disk, the next request's seek can be overlapped with the current request's bus transfer, yielding the full disk efficiency benefits shown in Figure 1. Fortunately, most modern disks and most current operating systems support command queueing at the disk.

4 Implementation

We have developed a prototype implementation of a traxtent-aware file system in FreeBSD. This implementation identifies track boundaries and modifies the FreeBSD FFS implementation to take advantage of this information. This section describes our algorithms for detecting track boundaries and details our modifications to FFS.

4.1 Detecting track boundaries

We have implemented two approaches to detecting track boundaries: a general approach applicable to any disk interface supporting a read command and a specialized approach for SCSI disks.

4.1.1 General approach

The general extraction algorithm locates track boundaries by identifying discontinuities in access efficiency. Recall from Figure 1 that disk efficiency for track-aligned requests increases linearly with the number of sectors being transferred until a track boundary is crossed. Starting with sector 0 of the disk ($S = 0$), the algorithm issues successive requests of increasing size, each starting at sector S (i.e., read 1 sector starting at S, read 2 sectors starting at S, etc.). The extractor avoids rotational latency variance by synchronizing with the rotation speed, issuing each request at (nearly) the same offset in the rotational period; rotational latency could also be addressed by averaging many observations, but at a substantial cost in extraction time. Eventually, an N-sector read returns in more time than a linear model suggests (i.e., $N = SPT + 1$), which identifies sector $S + N$ as the start of a new track. The algorithm then repeats with $S = S + N - 1$.

The method described above is clearly suboptimal; our actual implementation uses a binary search algorithm to find when $N = SPT + 1$. In addition, once SPT is determined for a track, the common case of each subsequent track being the same size is quickly verified. This verification checks for a discontinuity between $S + SPT - 1$ and $S + SPT$. If so, it sets $S = S + SPT - 1$ and moves on. Otherwise, it sets $N = 1$ and uses the base method; this occurs mainly on the first track of each zone and on tracks containing defects. With these enhancements, the algorithm extracts the track boundaries of a 9 GB disk (a Quantum Atlas 10K) in four hours. Talagala et al. [44] describe a much quicker algorithm that extracts approximate geometry information using just the read command; however, for our purposes, the exact track boundaries must be identified.

One difficulty with using read requests to detect track boundaries is the caching performed by disk firmware. To obviate the effects of firmware caching, the algorithm interleaves 100 parallel extraction operations to widespread disk locations, such that the cache is flushed each time we return to block S. An alternative approach would be to use write requests; however, this is undesirable because of the destructive nature of writes and because some disks employ write-back caching.

4.1.2 SCSI-specific approach

The SCSI command set supports query operations that can simplify track boundary detection. Worthington et al. [48] describe how these operations can be used to determine LBN-to-physical mappings. Building upon their basic mechanisms, we have implemented an automated disk drive characterization tool called DIXtrac [35]. This tool includes a five-step algorithm that exploits the regularity of disk geometry and layout characteristics to efficiently and automatically extract the complete LBN-to-physical mappings in less than one minute (fewer than 30,000 LBN translations), largely independent of disk capacity:

1. Use the READ CAPACITY command to determine the highest LBN, and determine the number of cylinders and surfaces by mapping random and targeted LBNs to physical locations using the SEND/RECEIVE DIAGNOSTIC command.
2. Use the READ DEFECT LIST command to obtain a list of all media defect locations.
3. Determine where spare sectors are located on each track and cylinder, and detect any other space reserved by the firmware. This is done by an expert-system-like process of combining the results of several queries, including whether or not (a) each track in a cylinder has the same number of LBN-holding sectors; (b) one cylinder within a set has fewer sectors than can be explained by the defect list; and (c) the last cylinder in a zone has too few sectors.
4. Determine zone boundaries and the number of sectors per track in each zone by counting the sectors on a defect-free, spare-free track in each zone.
5. Identify the remapping mechanism used for each defective sector. This is determined by back-translating the LBNs returned in step 2.

DIXtrac has been successfully used on dozens of disks, spanning 11 different disk models from 4 different manufacturers. Still, it does not always work. In our experience, step #3 has failed several times when we tried a new disk with a previously unknown (to us) mapping scheme — most are now part of DIXtrac's expertise, but future advances may again baffle it. When this happens, a system can fall back on the general approach or, better yet, a SCSI-specific version of it. That is, the general algorithm can be specialized to use SCSI's SEND/RECEIVE DIAGNOSTIC command instead of request timings. Such expertise-free, SCSI-specific extraction of track boundaries requires approximately 2.0–2.3 translations per track for most disks; it requires approximately 5 minutes for the 9GB Atlas 10K.

4.2 Traxtent support in FreeBSD

This section reviews the basic operation of FreeBSD FFS [25] and describes our changes to implement traxtent-aware allocation and access in FreeBSD.

4.2.1 FreeBSD FFS overview

FreeBSD assigns three identifying block numbers to buffered disk data (Figure 4). The `lblkno` represents the offset within a file; that is, the buffer containing the first byte of file data is identified by `lblkno` 0. Each `lblkno` is associated with one `blkno` (physical block number), which is an abstract representation of disk addresses used by the OS to simplify space management. Each `blkno` directly maps to a range of contiguous disk *sector numbers* (LBNs), which are the actual addresses presented to the device driver during an access. (Device drivers adjust sector numbers to partition boundaries.) In our experiments, the file system block size is 8 KB (sixteen contiguous LBNs). In this section, "block" refers to a physical block.

FFS partitions the set of physical blocks into fixed-size block groups ("cylinder groups"). Each block group contains a small amount of summary information—inodes, free block map, etc.—followed by a large contiguous array of data blocks. Block group size, block allocation, and media access characteristics were once based on the

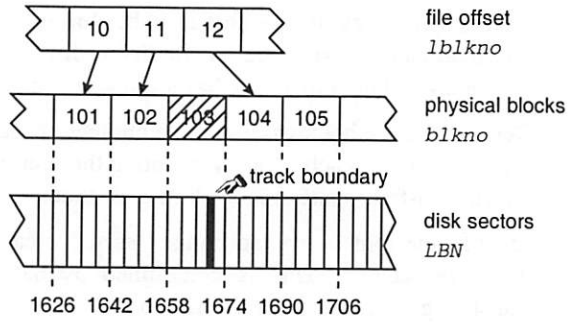


Figure 4: **Mapping system-level blocks to disk sectors.** Physical block 101 maps directly to disk sectors 1626–1641. Block 103 is an *excluded* block (see Section 4.2.2) because it spans the disk track boundary between LBNs 1669–1670.

underlying disk's physical geometry. Although this geometry dependence is no longer real, block groups are still used in their original form because they localize related data (e.g., files in the same directory) and their inodes, resulting in more efficient disk access. The block groups created for our experiments are 32 MB in size.

FreeBSD's FFS implementation uses the clustered allocation and access algorithms described by McVoy & Kleiman [26]. When newly created data are committed to disk, blocks are allocated to a file by selecting the closest "cluster" of free blocks (relative to the last block committed) large enough to store all N blocks of buffered data. Usually, the cluster selected consists of the N blocks immediately following the last block committed. To assist in fair local allocation among multiple files, FFS allows only half of the blocks in a block group to be allocated to a single file before switching to a new block group.

FFS implements a history-based read-ahead (a.k.a. prefetching) algorithm when reading large files sequentially. The system maintains a "sequential count" of the last run of sequentially accessed blocks (if the last four accesses were for blocks 17, 20, 21, and 22, the sequential count is 3). When the number of cached read-ahead blocks drops below 32, FFS issues a new read-ahead of length l beginning with the first noncached block, where l is the lowest of (a) the sequential count, (b) the number of contiguously allocated blocks remaining in the current cluster, or (c) 32 blocks[1].

4.2.2 FreeBSD FFS modifications

This section describes the few, small changes required to integrate traxtent-awareness into FreeBSD FFS.

[1] 32 blocks is a representative default value. It may be smaller on systems with limited resources or larger on systems with custom kernels.

Excluded blocks and traxtent allocation. We introduce the concept of the *excluded* block, highlighted in Figure 4. Blocks that span track boundaries are excluded from allocation decisions by marking them as used in the free-block map. Whenever the preferred block (the next sequential block) is excluded, we instead allocate the first block of the closest available traxtent. When possible, mid-size files are allocated such that they fit within a single traxtent. On average, one out of every twenty blocks of the Quantum Atlas 10K is excluded under our modified FFS. As per-track capacity grows, the frequency of excluded blocks decreases—for the Atlas 10K II, one in thirty is excluded.

Traxtent-sized access. No fundamental changes are necessary in the FFS clustered read-ahead algorithm. FFS properly identifies runs of blocks between excluded blocks as clusters and accesses them with a single disk request. Until non-sequential access is detected, we ignore the "sequential count" to prevent multiple partial accesses to a single traxtent; for non-sequential file sessions, the default mechanism is used. We handle the special case where there is no excluded block between traxtents by ensuring that no read-ahead request goes beyond a track boundary. At a low level, unmodified FreeBSD already supports command queuing at the device and attempts to have at least one outstanding request for each active data stream.

Traxtent data structures. When the file system is created, track boundaries are identified, adjusted to the file system's partition, and stored on disk. At mount time, they are read into an extended FreeBSD `mount` structure. We chose the mount structure because it is available everywhere traxtent information is needed.

5 Evaluating Traxtents

This section examines the performance benefits of track-based access at two levels. First, it evaluates the disk in isolation, finding a 50% improvement in disk efficiency and a reduction in response time variance. Second, it quantifies system-level performance gains, showing a 20% reduction in run time for large file operations, a 56% increase in the number of concurrent streams serviceable on a video server, and a 44% lower write cost for a log-structured file system.

5.1 Experimental setup

Most experiments described in this section were performed on two disks that support zero-latency access (Quantum Atlas 10K and Quantum Atlas 10K II) and two disks that do not (Seagate Cheetah X15 and IBM Ultrastar 18 ES). The disks were attached to a 550 MHz Pentium III-based PC. The Atlas 10K II was attached via

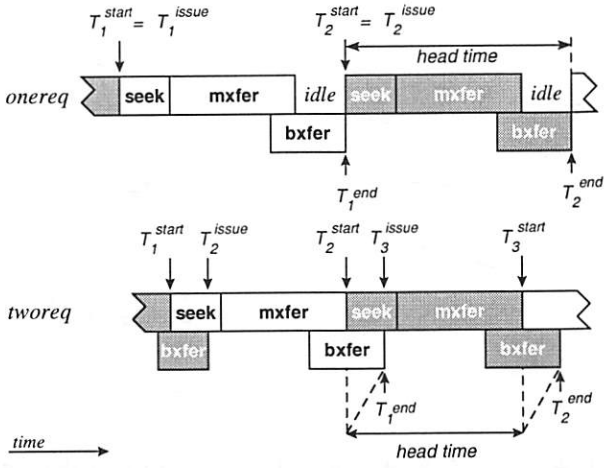


Figure 5: **Expressing head time.** The head time of a *onereq* request is $T_2^{end} - T_2^{issue}$. For *tworeq*, the head time is $T_2^{end} - T_1^{end}$. T^{issue} is the time when the request is issued to the disk, T^{start} is when the disk starts servicing the request, and T^{end} is when completion is reported. Notice that for *tworeq*, T^{issue} does not equal T^{start} because of queueing at the disk.

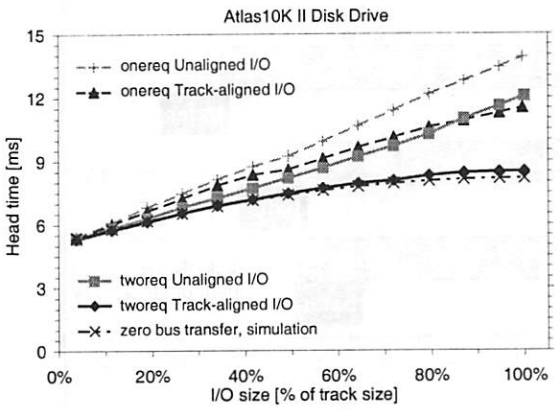


Figure 6: **Average head time for track-aligned and unaligned reads for Quantum Atlas 10K II.** The dashed and solid lines show the average of measured times for 5000 random track-aligned and unaligned reads to the disk's first zone for the *onereq* and *tworeq* workloads. Multiple runs for a subset of the points reveal little variation (<0.4%) between average head times for distinct sets of 5000 random requests. The thin dotted line represents the *onereq* workload replayed on a simulator configured with zero bus transfer time; note that it approximates *tworeq* without having to ensure queued requests at the disk.

an Adaptec Ultra160 Wide SCSI adapter, the Atlas 10K and Ultrastar were attached via an 80 MB/s Ultra2 Wide SCSI adapter, and the Cheetah via a Qlogic FibreChannel adapter. We also examined workloads with the DiskSim disk simulator [16] configured to model the respective disks. Examining these disks in simulation enables us to quantify the individual components of the overall response time, such as seek and bus transfer time.

5.2 Disk performance

Two workloads, *onereq* and *tworeq*, are used to evaluate basic track-aligned performance. Each workload consists of 5000 random requests within the first zone of the disk. The difference is that *onereq* keeps only one outstanding request at the disk, whereas *tworeq* ensures one request is always queued at the disk in addition to the one being serviced.

We compare the efficiency of both workloads by measuring the average per-request *head time*. A request's head time is the amount of time that the disk head is dedicated to that request. The average head time is the reciprocal of request throughput (i.e., I/Os per second). Therefore, higher disk efficiency will result in a shorter average head time, all else being equal. We introduce head time as a metric because it allows us to identify component delays more easily.

For *onereq* requests, head time equals disk response time as observed by the device driver, because the next request is not issued until the current one is complete. As usual, disk response time is the elapsed time from when a request is sent to the disk to when completion is reported. For *onereq* requests, the read/write head is idle for part of this time, because the only outstanding request is waiting for a bus transfer to complete. For *tworeq* requests, the head time includes only media access delays, since bus activity for any one request is overlapped with positioning and media access for another. The components of head times for the *onereq* and *tworeq* workloads are shown graphically in Figure 5.

Read performance. Figure 6 shows the improvement given by track-aligned accesses on the Atlas 10K II. For track-sized requests, head times for track-aligned accesses in *onereq* and *tworeq* decrease by 18% and 32% respectively, which correspond to increases of 22% and 47% in efficiency. The *tworeq* efficiency increase exceeds that of *onereq* because *tworeq* overlaps the previous request's bus transfer with the current request's media transfer.

Because bus and media transfers are overlapped, the head time for a track-aligned, track-sized request in the *tworeq* workload is 8.3 ms (calculated as shown in Figure 5). Subtracting 2.2 ms average seek time from the head time yields 6.1 ms. This observed value is very close to the

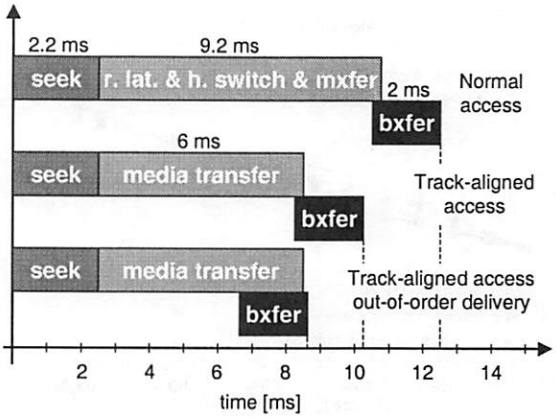


Figure 7: **Breakdown of measured response time for a zero-latency disk.** "Normal access" represents track-unaligned access, including seek, rotational latency (*r.lat.*), head switch, media transfer (*mxfer*), and bus transfer (*bxfer*). For track-aligned access, the in-order bus transfer does not overlap the media transfer. With out-of-order bus delivery, overlap of bus and media transfers is possible.

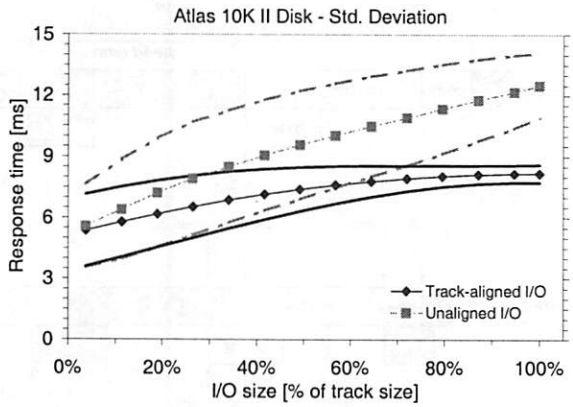


Figure 8: **Response time and its standard deviation for track-aligned and unaligned disk access.** The thin lines with markers represent the average response time, and the envelope of thick lines is the response time ± one standard deviation. The data shown in the graph was obtained by running the *onereq* workload on a simulated disk configured with an infinitely fast bus to eliminate the response time variance due to in-order bus delivery.

rotation time of 6 ms, confirming that track-aligned accesses to zero-latency disks can fetch a full track in one revolution with no rotational latency.

The command queueing of *tworeq* is needed in current systems to address the in-order bus delivery requirement. That is, even though zero-latency disks can read data out of order, they only send data over the bus in ascending LBN order. This results in only a 3% overlap, on average, between the media transfer and bus transfer for the track-aligned access bar in Figure 7. The overlap would be nearly complete if out-of-order bus delivery were used instead, as shown by the bottom bar. Out-of-order bus delivery would improve the efficiency of *onereq* to nearly that of *tworeq* while relaxing the queueing requirement (shown as the "zero bus transfer" curve in Figure 6). Although the SCSI specification allows out-of-order bus delivery using the MODIFY DATA POINTER command, we are not aware of any disks that support this operation.

Write performance. Track-alignment also makes writes more efficient. For the *onereq* workload on the Atlas 10K II, the head time of track-sized writes is 10.0 ms for track-aligned access and 13.9 ms for unaligned access, which is a reduction of 28%. For *tworeq*, the reduction in head time is 26% (from 13.8 ms to 10.2 ms). These reductions correspond to efficiency increases of 39% and 35%, respectively.

The larger *onereq* improvement, relative to reads, occurs because the seek and bus transfer are overlapped. The disk can initiate the seek as soon as the write command arrives. While the seek is in progress, the data is transferred to the disk and buffered. Since the average seek for the *onereq* workload is 2.2 ms and the data transfer takes about 2 ms, the data usually arrives at the disk before the seek is complete and the zero-latency write begins.

Importance of zero-latency access. The head time reductions for the other zero-latency disk (the Atlas 10K) are 16% and 32% for track-sized reads in the *onereq* and *tworeq* workloads, corresponding to 19% and 47% higher efficiencies. These reductions are smaller due to the Atlas 10K's longer average seek time of 2.4 ms.

Head time does not drop as significantly for track-aligned reads on disks that do not support zero-latency access: 6% for the IBM Ultrastar 18ES and 8% for the Seagate Cheetah X15. For these disks, aligning accesses on track boundaries only eliminates the 0.8–1.1ms head switch time—the rotational latencies of 4 ms (Ultrastar) and 2 ms (Cheetah) are still incurred.

Response time variance. Track-aligned access can significantly lower the standard deviation, σ, of response time as seen in Figure 8. As the request size increases from one sector to the track size, $\sigma_{aligned}$ decreases from 1.8 ms to 0.4 ms, whereas $\sigma_{unaligned}$ decreases from 2.0 ms to 1.5 ms. The standard deviation of the seeks in this workload is 0.4 ms, indicating that the response time variance for aligned access is due entirely to the seeks. Lower variance makes response times more predictable, allowing soft real-time applications to use tighter bounds in scheduling and thereby achieve higher utilization.

Track-based requests also have lower worst-case access times, since rotational latency and head switch time are avoided.

5.3 FFS experiments

Building on the disk-level results, this section compares our prototype traxtent-aware FFS to unmodified FFS. We also include results for a modified FFS, here called *fast start* FFS, that aggressively prefetches contiguous blocks. The unmodified FFS slowly ramps up its prefetching as it observes sequential access to a file. The fast start FFS, on the other hand, prefetches up to 32 contiguous blocks on the first access to a file, thus approximating the behavior of the traxtent-aware FFS (albeit with larger requests and no knowledge of track boundaries).

Each test is performed on a freshly-booted system with a clean partition on a Quantum Atlas 10K. The tests verify the expected performance effects: small penalty for single sequential scan, substantial benefit for interleaved scans, and no effect on small file activity. We also identify and measure the worst-case scenario. The results are summarized in Table 2.

Single large file. The first experiment is an I/O-bound linear scan through a 4 GB file. As expected, traxtent-FFS runs 5% slower than unmodified FFS or fast start FFS (199.8 s vs. 189.6 s and 188.9 s respectively). This is because FFS is optimized for large sequential single-file access and reads at the maximum disk streaming rate, whereas traxtent-FFS inserts an excluded block one out of every twenty blocks (5%). This penalty could be eliminated by changing the file system cache to support buffering of partial blocks (much like IP fragments) instead of using excluded blocks in large files; this approach would give the block-based system extent-like flexibility.

Multiple large files. The second experiment consists of the `diff` application comparing two large files. Because `diff` interleaves fetches from the two files, we expect to see a speedup from improved disk efficiency. For 512 MB files, traxtent-FFS completes 19% faster than unmodified FFS or fast start FFS. A more detailed analysis shows that traxtent-FFS performs 6724 I/Os (average size of 160 KB) in 56.6 s while unmodified FFS performs only 4108 I/Os (mostly 256 KB) but requires 69.7 s. The fast start FFS performs 4094 I/Os (all but one at 256 KB) and requires 70.0 s. Subtracting media transfer time, unmodified FFS incurs 6.9 ms of overhead (seek + rotational latency + track switch time) per request, and traxtent-FFS incurs only 2.2 ms of overhead per request. In fact, the 19% improvement in overall completion time corresponds to an improvement in disk efficiency of 23%, exactly matching the predicted difference between single-track accesses and 256 KB unaligned accesses on an Atlas 10K disk.

The third experiment verifies write performance by copying a 1 GB file to another file in the same directory. FFS commits dirty buffers as soon as a complete cluster is created, which results in two interleaved request streams to the disk. This test shows a 20% reduction in run time for traxtent-FFS over unmodified FFS (124.9 s vs. 156.9 s). The fast start FFS finished in 155.3 s.

Small Files. Two application benchmarks are used to verify that the traxtent modifications do not penalize small file workloads. Postmark [21] simulates the small-file activity of busy Internet servers. Our experiments use Postmark v1.11 and its default parameters: 5–10KB files and 1:1 read-to-write and create-to-delete ratios. SSH-build [38] represents software development activity, replacing the Andrew benchmark. Its three phases unpack the compressed tar archive of SSH v1.2.27, generate the header files and Makefiles, and build the program executable.

As expected, we observe little difference. The SSH-build results differ by less than 0.2%, because the file system activity is dominated by small synchronous writes and cache hits. The fast start FFS performs exactly like the traxtent FFS having an edge of 0.2% over the unmodified FFS. Postmark is 4% faster with traxtents (55 transactions/second versus 53 for both unmodified and fast start FFS), because the few track switches are avoided. Fast start is not important for Postmark, because the files consist of only 1–3 blocks.

One might view these results as a negative indication of traxtents' value, but they are not. Recall that FreeBSD FFS does not explicitly group small files into large disk requests. Such grouping has been shown to yield 2–8× throughput increases for static web servers [20], web proxy caches [39], and software development activities [15]. Based on our measurements, we expect that the additional 50% increase in throughput from traxtents would be realized given such grouping.

Worst case scenario. As expected, we observe no penalty to small file I/O and a minimal (5%) penalty to the unoptimized single stream case. For random file I/O, FFS's "sequential count" prefetch control replaces the traxtent-based fetch mechanism, preventing useless full-track reads. The one remaining worst-case scenario would be single-block reads to the beginnings of many large files; in this case, the original FFS will fetch the first 8KB block and prefetch the second, whereas the modified FFS will fetch the entire first traxtent (≈ 160 KB). To evaluate this scenario, we ran an experiment, called `head *`, that reads the first byte of 1000 200 KB files. The results show a 45% penalty for traxtents (3.6 s vs.

	4GB `scan`	512MB `diff`	1GB `copy`	Postmark	SSH-build	`head *`
unmodified	189.6 s	69.7 s	156.9 s	53 tr/s	72.0 s	4.6 s
fast start	188.9 s	70.0 s	155.3 s	53 tr/s	71.5 s	5.5 s
traxtents	199.8 s	56.6 s	124.9 s	55 tr/s	71.5 s	5.2 s

Table 2: **FreeBSD FFS results.** All but the `head *` values are an average of three runs. The individual run times deviate from their average by less than 1%. The `head *` value is an average of five runs and the individual runs deviate by less than 3.5%. Postmark reported the same number of transactions per second in all three runs for the respective FFS, except for one run of the unmodified FFS that reported 54 transactions per second.

5.2 s), closely matching the predicted per-request service time difference (5.6 ms vs. 8.0 ms). Fortunately, this scenario is not often expected to arise in practice. Not surprisingly, the fast start FFS performs even worse than the traxtent FFS with an average runtime of 5.5 s as it prefetches even more unnecessary data.

5.4 Video servers

A video server is designed to serve large numbers of video streams to clients at guaranteed rates. To accomplish this, the server first fetches one time interval of video (e.g., 0.5 s) for each stream. This set of fetches is called a *round*. Then, while the data are transferred to clients from the server's buffers, the server schedules the next round of requests. Since the per-interval disk access time is less than the round time, many concurrent streams can be supported by a single disk. Further, by spreading video streams across D disks, D times as many concurrent streams can be supported.

The per-interval disk request size, *IOsize*, represents a trade-off between throughput (the number of concurrent streams) and other considerations (buffer space and start-up latency). *IOsize* must be large enough so that achieved disk bandwidth (disk efficiency times peak bandwidth) exceeds V times the video bit rate, where V is the number of concurrent video streams supported. As *IOsize* increases, both disk efficiency and $Time_{perIO}$ increase, increasing both the number of video streams that can be supported and the round time, which is defined as V times $Time_{perIO}$.

Round time determines the startup latency of a newly admitted stream. Assuming the video server spreads data across D disks, the worst-case startup latency is round time times $(D+1)$ [34]. The buffer space required at the server is $2 \times IOsize_{\text{disk}} \times V$. In practice, *IOsize* is chosen to meet system goals given a trade-off between startup latency and the maximum number of supportable streams. Since track-aligned access increases disk efficiency, it enables more concurrent streams to be serviced at a given *IOsize*.

5.4.1 Soft real-time

Most video server projects, such as Tiger [3] and RIO [34], provide soft real-time guarantees. These systems guarantee that, with a certain probability, a request will not miss its deadline. This allows a relaxation on the assumed worst-case seek and rotational latency and results in higher bandwidth utilization for both track-aligned and unaligned access.

We evaluate two video servers (one traxtent-aware and one not), each containing 10 Quantum Atlas 10K II disks, using the same approach as the RIO video server [34]. First, we measured the time to complete a given number of simultaneous, random track-sized requests. This measurement was repeated 10,000 times for each number of simultaneous requests from 10 to 80. (80 is the maximum number of simultaneous 4 Mb/s streams that can be supported by each disk's 40 MB/s streaming bandwidth.)

From the PDF of the measured response times, we obtained the round time that would meet 99.99% of the deadlines for the 4 Mb/s rate. Given a 0.5 s round time (which translates to a worst-case startup latency of 5.5 s for the 10-disk array), the track-aligned system can support up to 70 streams per disk. In contrast, the unaligned system is only able to support 45 streams per disk. Thus, the track-aligned system can support 56% more streams at this minimal startup latency.

To support more than 70 and 45 streams per disk for the track-aligned and unaligned systems, the I/O size must increase. This increase in I/O size causes an increase in the round time, which in turn increases the startup latency as shown in Figure 9. At 70 streams per disk, the startup latency for the track-aligned system is 4× smaller than for the track-unaligned system.

5.4.2 Hard real-time

Although many video servers implement soft real-time requirements, there are applications that require hard real-time guarantees. In their admission control algorithms, these systems must assume the worst-case response time to ensure that no deadline is missed. In

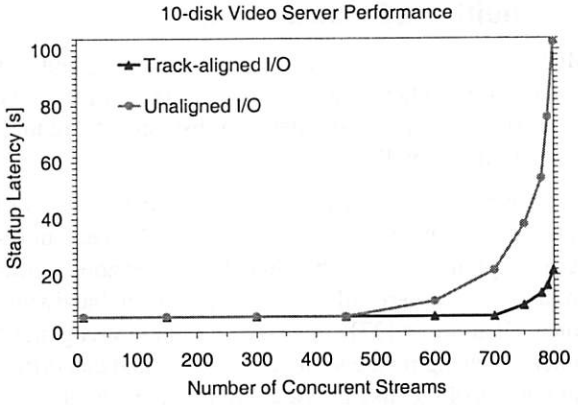


Figure 9: **Worst-case startup latency of a video stream for track-aligned and unaligned accesses.** The startup latency is shown for a 10-disk array of Quantum Atlas 10K II disks, which can support up to 800 concurrent streams.

computing the worst-case response time, one assumes the worst-case seek, transfer time, and rotational latency. Both the track-aligned and unaligned systems have the same values for the worst-case seek[2]. However, the worst-case rotational latency for unaligned access is one revolution, whereas track-based access suffers no rotational latency. The worst-case transfer time will be similar except that the unaligned system must assume at least one head switch will occur for each request. With a 4 Mb/s bit rate and an I/O size of 264 KB, the track-unaligned system supports 36 streams per disk whereas the track-based system supports up to 67 streams. This translates into 45% and 83% disk efficiency, respectively. With an I/O size of 528 KB, unaligned access yields 52 streams vs. 75 for track-based access. Unaligned I/O size must exceed 2.5 MB, with a maximum startup latency of 60.5 seconds, to achieve the same efficiency as the track-aligned system.

5.5 Log-structured File System

The log-structured file system (LFS) [33] was designed to reduce the cost of disk writes. Towards this end, it remaps all new versions of data into large, contiguous regions called segments. Each segment is written to disk with a single I/O operation, amortizing the positioning cost over one large write. A significant challenge for LFS is ensuring that empty segments are always available for new data. LFS answers this challenge with an internal defragmentation operation called *cleaning*. Cleaning of a previously written segment involves identifying the subset of "live" blocks, reading them into memory, and writing them into a new segment. Live blocks are those that have not been overwritten or deleted by later operations.

There is a performance trade-off between write efficiency and the cost of cleaning. Larger segments offer higher write efficiency but incur larger cleaning cost since more data has to be transferred for cleaning [24, 37]. Additionally, the transfer of large segments hurts the performance of small synchronous reads [5, 24]. Given these conflicting pressures, the choice of segment size must balance write efficiency, cleaning cost, and small synchronous I/O performance. Matching segments to track boundaries can yield higher write efficiency with smaller segments and thus lower cleaning costs.

To evaluate the benefit of using track-based access for LFS segments, we use the overall write cost (*OWC*) metric described by Matthews et al. [24], which is a refinement of the *write cost* metric defined for the Sprite implementation of LFS [33]. It expresses the cost of writes in the file system, assuming that all data reads are serviced from the system cache. The *OWC* metric is defined as the product of write cost and disk transfer inefficiency:

$$\begin{aligned} OWC &= WriteCost \times TransferInefficiency \\ &= \frac{N^{new}_{written} + N^{clean}_{read} + N^{clean}_{written}}{N^{data}_{written}} \times \frac{T^{actual}_{xfer}}{T^{ideal}_{xfer}} \end{aligned}$$

where N is the number of segments written due to new data or read and written due to segment cleaning, and T is the time for one segment transfer. *WriteCost* depends on the workload (i.e., how much new data is written and how much old data is cleaned) but is independent of disk characteristics. *TransferInefficiency*, on the other hand, depends only on disk characteristics. Therefore, we can use the *WriteCost* values given by Matthews et al. for their Auspex server trace [24] and measured *TransferInefficiency* values like those in Figure 1.

Figure 10 shows that *OWC* is lower with track-aligned disk access and that the cost is minimized when the segment size matches the track size. Unlike our use of empirical data for determining *TransferInefficiency*, Matthews et al. estimate its value as

$$TransferInefficiency = T_{pos} \times \frac{BW_{disk}}{S_{segment}} + 1$$

where $S_{segment}$ is the segment size (in bytes) and T_{pos} is the average positioning time (i.e., seek and rotational latency). To verify that our results are in agreement with their findings, we computed *OWC* for the Atlas 10K II based on its specifications and plotted it in Figure 10 (labeled "5.2 ms*40 MB/s") with the *OWC* values for the

[2]The worst-case time for V seeks is much smaller than V times a full strobe seek (seek from one edge of the disk to the other), decreasing with increasing number (V) of concurrent streams [31]. This is because the disk scheduler can sort the requests in each round to minimize total seek distance. The worst-case seek time charged to a stream is equal to the worst-case scheduled seek route that serves all streams divided by the number of streams.

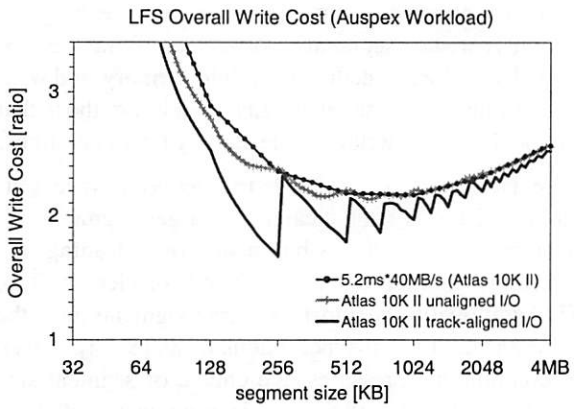


Figure 10: **LFS overall write cost for the Auspex trace as a function of segment size.** The line labeled "5.2 ms*40 MB/s" is the overall write cost predicted by the transfer inefficiency model described by Matthews et al. [24].

track-aligned and unaligned I/O. Because the empirical values are for the disk's first zone, the model values are too: 2.2 ms average seek, 3 ms average rotational latency, and peak bandwidth of 40 MB/s. As expected, the model is a good match for the unaligned case.

5.5.1 Variable segment size

As shown in Figure 10, the lowest write cost is achieved when the size of a segment matches the size of a track. However, different tracks may hold different numbers of LBNs. Therefore, an LFS must allow for variable segment sizes in order to match segment boundaries to track boundaries. Fortunately, doing so is straightforward.

In an LFS, the segment usage table records information about each segment. In the SpriteLFS implementation [33], this table is kept as an in-memory kernel structure and is stored in the checkpoint region of the file system. The BSD-LFS implementation [36] stores this table in a special file called the IFILE. Because of its frequent use, this file is almost always in the file system's cache.

Variable-sized segments can be supported by augmenting the per-segment information in the segment usage table with a starting location (the LBN) and length. During the initialization, each segment's starting location and length are set according to the corresponding track boundary information. When a new segment is allocated in memory, its size is looked up in the segment usage table. When the segment becomes full, it is written to the disk at the starting location given in the segment usage table. The procedures for reading segments and for cleaning are similar.

6 Additional Related Work

Much related work has been discussed throughout this paper. Some other notable related work has promoted zone-based allocation and detailed disk-specific request generation for small requests.

The Tiger video server [3] allocated primary copies of videos to the outer portions of disks' LBN space in order to exploit the higher bandwidth of outer zones. Secondary copies were allocated to the lower bandwidth zones. Van Meter [27] suggested that there was general benefit in changing file systems to understand that different regions of the disk provide different bandwidths.

By utilizing even more detailed disk information, several researchers have shown substantial decreases in small request response times [8, 10, 13, 46, 49]. For small writes, these systems detect the position of the head and re-map data to the nearest free block in order to minimize the positioning costs [10, 46]. For small reads, the SR-Array [49] determines the head position when the read request is to be serviced and reads the closest of several replicas.

7 Summary

This paper presents a case for track-aligned extents. It demonstrates feasibility with a working prototype, and it demonstrates value with direct measurements. At the low level, traxtent accesses are shown to increase disk efficiency by approximately 50% compared to track-unaligned accesses of the same size. At the system level, traxtents are shown to increase application efficiency by 25–56% for large file workloads, video servers, and write-bound log-structured file systems.

Acknowledgements

Bill Nace contributed to our initial investigation into track-aligned extents. We thank our shepherd John Wilkes and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research effots. John Griffin is funded in part by a National Science Foundation Graduate Fellowship.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA).

Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991.

[2] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 277–288. USENIX Association, 1995.

[3] William J. Bolosky, Joseph S. Barrera, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid. *The Tiger video fileserver*. Technical Report MSR–TR–96–09. Microsoft Corporation, April 1996.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, 2001.

[5] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 79–91, 21–22 May 1992.

[6] Edward Chang and Hector Garcia-Molina. Reducing initial latency in a multimedia storage system. *International Workshop on Multi-Media Database Management Systems* (Blue Mountain Lake, NY), pages 2–11, 14–16 August 1996.

[7] Edward Chang and Hector Garcia-Molina. Effective memory use in a media server. *VLDB* (Athens, Greece.), pages 496–505, 26–29 August 1997.

[8] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. *Mime: high performance parallel storage device with strong recovery guarantees*. Technical report HPL-92-9. 18 March 1992.

[9] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.

[10] Tzi-Cker Chiueh and Lan Huang. *Trail: track-based logging in Stony Brook Linux*. Technical report ECSL TR-68. SUNY Stony Brook, December 1999.

[11] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: a new approach to improving file systems. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 15–28, 5–8 December 1993.

[12] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Alanta, Georgia), pages 59–70, 1–4 May 1999.

[13] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Winter USENIX Technical Conference* (San Francisco, CA), pages 237–251. Usenix, 20–24 January 1992.

[14] Eran Gabber and Elizabeth Shriver. Lets put NetApp and CacheFlow out of business. *SIGOPS European Workshop* (Kolding, Denmark), pages 85–90, 17–20 Sept. 2000.

[15] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *Annual USENIX Technical Conference* (Anaheim, CA), pages 1–17, January 1997.

[16] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim simulation environment version 1.0 reference manual*, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[17] Sanjay Ghemawat. *The modified object buffer: a storage management technique for object-oriented databases*. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, 7 September 1995.

[18] Dominic Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann, 1998.

[19] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[20] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. *ACM SIGOPS. European workshop: Systems support for worldwide applications* (Connemara, Ireland, September 1996), pages 141–148. ACM, 1996.

[21] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[22] Kimberly Keeton and Randy H. Katz. The evaluations of video layout strategies on a high-bandwidth file server. *4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Lancaster, England, UK.), pages 228–229, 3–5 November 1993.

[23] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[24] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[25] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[26] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Winter USENIX Technical Conference* (Dallas, TX), pages 33–43, 21–25 January 1991.

[27] Rodney Van Meter. Observing the effects of multi-zone disks. *Annual USENIX Technical Conference* (Anaheim, CA), pages 19–30, 6–10 January 1997.

[28] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, 1997.

[29] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson.

A trace-driven analysis of the UNIX 4.2 BSD file system. *ACM Symposium on Operating System Principles* (Orcas Island, WA). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[30] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB SCSI product manual*, Document number 81-119313-05, August 1999.

[31] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. *International Multimedia Conference* (Anaheim, CA, 01–06 August 1993), pages 225–234. ACM Press, 1993.

[32] ReiserFS. http://www.namesys.com/.

[33] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[34] Jose Renato Santos, Richard R. Muntz, and Berthier Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. *ACM SIGMETRICS 2000* (Santa Clara, CA). Published as *Performance Evaluation Review*, **28**(1):44–55, 17–21 June 2000.

[35] Jiri Schindler and Gregory R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[36] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 307–326, January 1993.

[37] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.

[38] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA), 18–23 June 2000.

[39] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage management for web proxies. *Annual USENIX Technical Conference* (Boston, MA, 25–30 June 2001), pages 203–216, 2001.

[40] Tracy F. Sienknecht, Rich J. Friedrich, Joe J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, **20**(1–3):3–25, May 1994.

[41] Keith A. Smith and Margo Seltzer. A comparison of FFS disk allocation policies. *USENIX.96* (San Diego, CA., 22–26 January 1996), pages 15–25. USENIX Assoc., 1996.

[42] Iceberg 9200 disk array subsystem. Storage Technology Corporation, 2270 South 88th Street, Louisville, CO 80028-4358, 9 June 1995.

[43] Adam Sweeney. Scalability in the XFS file system. *USENIX*. (San Diego, California), pages 1–14, 22–26 January 1996.

[44] Nisha Talagala, Remzi H. Dusseau, and David Patterson. *Microbenchmark-based extraction of local and global disk characteristics*. Technical report CSD–99–1063. University of California at Berkeley, 13 June 2000.

[45] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[46] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, 1999.

[47] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[48] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.

[49] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.

Freeblock Scheduling Outside of Disk Firmware

Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger
Carnegie Mellon University

Abstract

Freeblock scheduling replaces a disk drive's rotational latency delays with useful background media transfers, potentially allowing background disk I/O to occur with no impact on foreground service times. To do so, a freeblock scheduler must be able to very accurately predict the service time components of any given disk request — the necessary accuracy was not previously considered achievable outside of disk firmware. This paper describes the design and implementation of a working external freeblock scheduler running either as a user-level application atop Linux or inside the FreeBSD kernel. This freeblock scheduler can give 15% of a disk's potential bandwidth (over 3.1MB/s) to a background disk scanning task with almost no impact (less than 2%) on the foreground request response times. This can increase disk bandwidth utilization by over 6×.

1 Introduction

Freeblock scheduling is an exciting new approach to utilizing more of a disk's potential media bandwidth. It consists of anticipating rotational latency delays and filling them with media transfers for background tasks. Via simulation, our prior work [14] indicated that 20–50% of a never-idle disk's bandwidth could be provided to background applications with no effect on foreground response times. This *free bandwidth* was shown to enable free segment cleaning in a busy log-structured file system (LFS), or free disk scans (e.g., for data mining or disk media scrubbing) in an active transaction processing system.

At the time of that writing, we and others believed that freeblock scheduling could only be done effectively from inside the disk's firmware. In particular, we did not believe that sufficient service time prediction accuracy could be achieved from outside the disk. We were wrong.

This paper describes and evaluates working prototypes of freeblock scheduling on Linux and within the FreeBSD kernel. Recent research has successfully demonstrated software-only Shortest-Positioning-Time-First (SPTF) [12, 25] schedulers [28, 31], but their prediction accuracies were not high enough to support freeblock scheduling. To squeeze extra media transfers into rotational latency gaps, a freeblock scheduler must be able to predict access times to within 200–300μs. It must also be able to deal with the drive's cache prefetching algorithms, since the most efficient use of a free bandwidth opportunity is on the same track as a foreground request.

These requirements can be met with two extensions to the common external SPTF design: limited command queueing and request merging. First, by keeping two requests outstanding at all times, an external scheduler can focus on just media access delays; the disk's firmware will overlap bus and command processing overheads for any one request with the media access of another. This tighter focus simplifies the scheduler's timing predictions, allowing it to achieve the necessary accuracy. Second, by merging physically adjacent free bandwidth and foreground fetches into a single request, an external scheduler can employ same-track fetches without confusing the firmware's prefetching algorithms.

With its service time prediction accuracy, our external scheduler's SPTF decisions match those of the disk's firmware, and its freeblock scheduling decisions are effective. On the other hand, the achieved free bandwidth is 35% lower than the earlier simulations, because the external prediction accuracies and control are not perfect. Nonetheless, the goals of freeblock scheduling are met: potential free bandwidth is used for background activities with (almost) no impact on foreground response times. For example, when using free bandwidth to scan the entire disk during on-line transaction processing, we measure 3.1 MB/s of steady-state progress or 37 free scans per day on a 9 GB disk. When employing freeblock scheduling, foreground response times increase by less than 2%.

The remainder of this paper is organized as follows. Section 2 describes freeblock scheduling. Section 3 describes challenges involved with implementing freeblock scheduling outside of disk firmware. Section 4 describes our implementation. Section 5 evaluates our external freeblock scheduler. Section 6 discusses related work. Section 7 summarizes this paper's contributions.

2 Freeblock Scheduling

Current high-end disk drives offer media bandwidths in excess of 40 MB/s, and the recent rate of improvement in media bandwidth exceeds 40% per year. Unfortunately, mechanical positioning delays limit most systems to only 2–15% of the potential media bandwidth. We recently

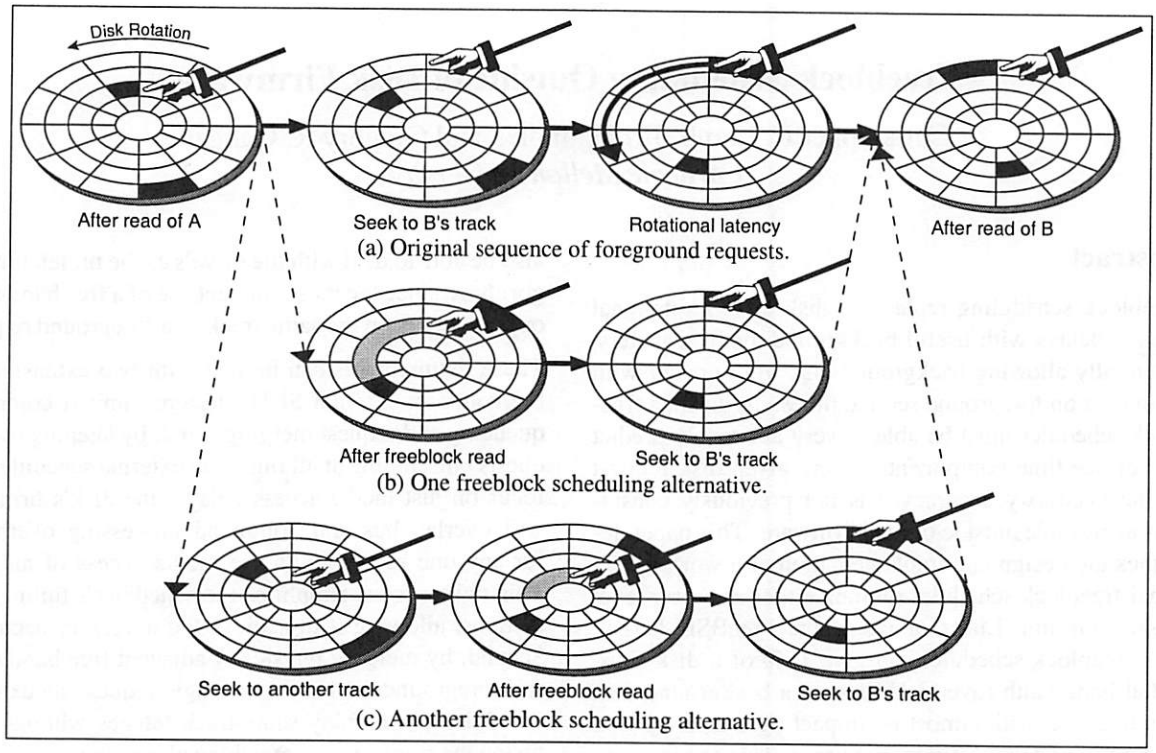


Figure 1: **Illustration of two freeblock scheduling possibilities.** Three sequences of steps are shown, each starting after completing the foreground request to block *A* and finishing after completing the foreground request to block *B*. Each step shows the position of the disk platter, the read/write head (shown by the pointer), and the two foreground requests (in black) after a partial rotation. The top row, labelled (a), shows the default sequence of disk head actions for servicing request *B*, which includes 4 sectors worth of potential free bandwidth (rotational latency). The second row, labelled (b), shows free reading of 4 blocks on *A*'s track using 100% of the potential free bandwidth. The third row, labelled (c), shows free reading of 3 blocks on another track, yielding 75% of the potential free bandwidth.

proposed freeblock scheduling as an approach to increasing media bandwidth utilization [14, 21]. By interleaving low-priority disk activity with the normal workload (here referred to as background and foreground, respectively), a freeblock scheduler can replace many foreground rotational latency delays with useful background media transfers. With appropriate freeblock scheduling, background tasks can make forward progress without any increase in foreground service times. Thus, the background disk activity is completed for free during the mechanical positioning for foreground requests.

This section describes the free bandwidth concept in greater detail, discusses how it can be used in systems, and outlines how a freeblock scheduler works. Most of the concepts were first described in our prior work [14] and are reviewed here for completeness.

2.1 Where the free bandwidth lives

At a high-level, the time required for a disk media access, T_{access}, can be computed as a sum of seek time, T_{seek}, rotational latency, T_{rotate}, and media access time, $T_{transfer}$:

$$T_{access} = T_{seek} + T_{rotate} + T_{transfer}$$

Of T_{access}, only the $T_{transfer}$ component represents useful utilization of the disk head. Unfortunately, the other two components usually dominate. While seeks are unavoidable costs associated with accessing desired data locations, rotational latency is an artifact of not doing something more useful with the disk head. Since disk platters rotate constantly, a given sector will rotate past the disk head at a given time, independent of what the disk head is doing up until that time. If that time can be predicted, there is an opportunity to do something more useful than just waiting for desired sectors to arrive at the disk head.

Freeblock scheduling is the process of identifying free bandwidth opportunities and matching them to pending background requests. It consists of predicting how much rotational latency will occur before the next foreground media transfer, squeezing some additional media transfers into that time, and still getting to the destination track in time for the foreground transfer. The additional media transfers may be on the current or destination tracks, on another track near the two, or anywhere between them, as illustrated in Figure 1. In the two latter cases, additional seek overheads are incurred, reducing the actual time available for the additional media trans-

fers, but not completely eliminating it.

The potential free bandwidth in a system is equal to the disk's potential media bandwidth multiplied by the fraction of time it spends on rotational latency delays. The amount of rotational latency depends on a number of disk, workload, and scheduling algorithm characteristics. For random small requests, about 33% of the total time is rotational latency for most disks. This percentage decreases with increasing request size, becoming 15% for 256 KB requests, because more time is spent on data transfer. This percentage increases with increasing locality, up to 60% when 70% of requests are in the most recent "cylinder group" [16], because less time is spent on the shorter seeks. The value is about 50% for seek-reducing scheduling algorithms (e.g., C-LOOK [17, 24] and Shortest-Seek-Time-First [9]) and about 20% for scheduling algorithms that reduce overall positioning time (e.g., Shortest-Positioning-Time-First).

2.2 Uses for free bandwidth

Potential free bandwidth exists in the time gaps that would otherwise be rotational latency delays for foreground requests. Therefore, freeblock scheduling must opportunistically match these potential free bandwidth sources to real bandwidth needs that can be met within the given time gaps. The tasks that will utilize the largest fraction of potential free bandwidth are those that provide the freeblock scheduler with the most flexibility. Tasks that best fit the freeblock scheduling model have low priority, large sets of desired blocks, and no particular order of access.

These characteristics are common to many disk-intensive background tasks that are designed to occur during otherwise idle time. For example, in many systems, there are a variety of support tasks that scan large portions of disk contents, such as report generation, RAID scrubbing, virus detection, and backup. Another set of examples is the many defragmentation [15, 29] and replication [18, 31] techniques that have been developed to improve the performance of future accesses. A third set of examples is anticipatory disk activities such as prefetching [7, 11, 13, 19, 27] and prewriting [2, 4, 8, 10].

Using simulation, our previous work explored two specific uses of freeblock scheduling. One set of experiments showed that cleaning in a log-structured file system [22] can be done for free even when there is no truly idle time, resulting in up to a 300% increase in application performance. A second set of experiments explored the use of free bandwidth for data mining on an active on-line transaction processing (OLTP) system, showing that over 47 full scans per day of a 9 GB disk can be made with no impact on OLTP performance. This resulted in a 7× increase in media bandwidth utilization.

2.3 Freeblock scheduling

In a system supporting freeblock scheduling, there are two types of requests: foreground requests and freeblock (background) requests. Foreground requests are the normal workload of the system, and they will receive top priority. Freeblock requests specify the background disk activity for which free bandwidth should be used. As an example, a freeblock request might specify that a range of 100,000 disk blocks be read, but in no particular order — as each block is retrieved, it is handed to the background task, processed immediately, and then discarded. A request of this sort gives the freeblock scheduler the flexibility it needs to effectively utilize free bandwidth opportunities.

Foreground and freeblock requests are kept in separate lists and scheduled separately. The foreground scheduler runs first, deciding which foreground request should be serviced next in the normal fashion. Any conventional scheduling algorithm can be used. Device driver schedulers usually employ seek-reducing algorithms, such as C-LOOK or Shortest-Seek-Time-First. Disk firmware schedulers usually employ Shortest-Positioning-Time-First (SPTF) algorithms [12, 25] to reduce overall positioning overheads (seek time plus rotational latency).

After the next foreground request (request *B* in Figure 1) is determined, the freeblock scheduler computes how much rotational latency would be incurred in servicing *B*; this is the free bandwidth opportunity. Like SPTF, this computation requires accurate estimates of disk geometry, current head position, seek times, and rotation speed. The freeblock scheduler then searches its list of pending freeblock requests for a good match. (Section 4.3 describes a specific freeblock scheduling algorithm.) After making its choice, the scheduler issues any free bandwidth accesses and then request *B*.

3 Fine-grain External Disk Scheduling

Fine-grain disk scheduling algorithms (e.g., Shortest-Positioning-Time-First and freeblock) must accurately predict the time that a request will take to complete. Inside disk firmware, the information needed to make such predictions is readily available. This is not the case outside the disk drive, such as in disk array firmware or OS device drivers.

Modern disk drives are complex systems, with finely-engineered mechanical components and substantial runtime systems. Behind standardized high-level interfaces, disk firmware algorithms map logical block numbers (LBNs) to physical sectors, prefetch and cache data, and schedule media and bus activity. These algorithms vary among disk models, and evolve from one disk generation to the next. External schedulers are isolated from

necessary details and control by the same high-level interfaces that allow firmware engineers to advance their algorithms while retaining compatibility. This section outlines major challenges involved with fine-grain external scheduling, the consequences of these challenges, and some solutions that mitigate the negative effects of these consequences.

3.1 Challenges

The challenges faced by a fine-grained external scheduler largely result from disks' high-level interfaces, which hide internal information and restrict external control. Specific challenges include coarse observations, non-constant delays, non-preemption, on-board caching, in-drive scheduling, computation of rotational offsets, and disk-internal activities.

Coarse observations. An external scheduler sees only the total response time for each request. These coarse observations complicate both the scheduler's initial configuration and its runtime operation. During initial configuration, the scheduler must deduce from these observations the individual component delays (e.g., mechanical positioning, data transfer, and command processing) as well as the amount of their overlap. These delays must be well understood for an external scheduler to accurately predict requests' expected response times. During runtime operation, the scheduler must deduce the disk's current state after each request; without this knowledge, the subsequent scheduling decision will be based on inaccurate information.

Non-constant delays. Deducing component delays from coarse observations is made particularly difficult by the inherent inter-request variation of those delays. If the delays were all constant, deduction could be based on solving sets of equations (response time observations) to figure out the unknowns (component delays). Instead, the delays and the amount of their overlap vary. As a result, an external scheduler must deduce moving targets (the component delays) from its coarse observations. In addition, the variation will affect response times of scheduled requests, and so it must be considered in making scheduling decisions. Figure 2 illustrates the effect of variable overlap between bus transfer and media transfer on the observed response time.

Non-preemption. Once a request is issued to the disk, the scheduler cannot change or abort it. The SCSI protocol does include an ABORT message, but most device drivers do not support it and disks do not implement it efficiently. They view it as an unexpected condition, so it is usually more efficient to just allow a request to complete. Thus, an external scheduler must take care in the decisions it makes.

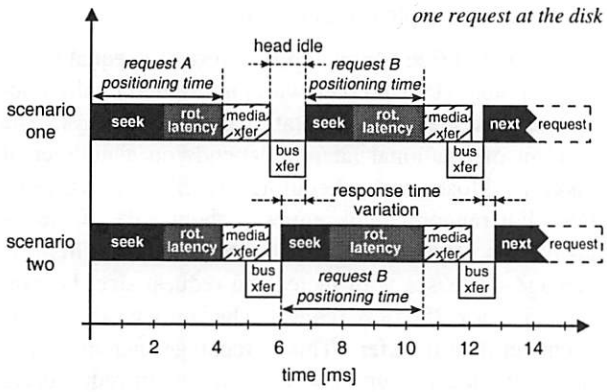


Figure 2: **Effects of uncertainty on prediction accuracy.** This figure shows two possible scenarios of observed response times when employing external scheduling. In each scenario, the scheduler issues request *A*, waits for its completion, and then issues request *B*. The two scenarios only differ in the amount of overlap between the media and bus transfers. The varying overlap has different effects on the positioning time of request B and therefore on the amount of available free bandwidth.

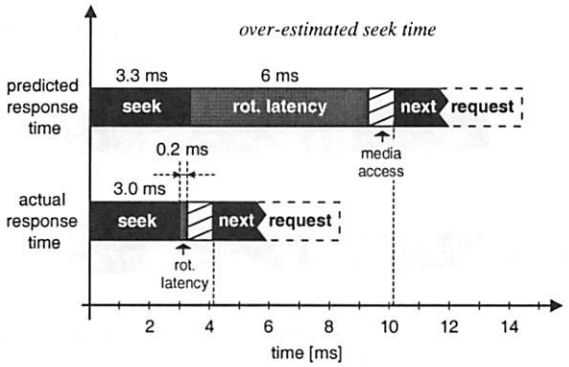
On-board caching. Modern disks have large on-board caches. Exploiting its local knowledge, disk firmware prefetches sectors into this cache based on physical locality. Usually, the prefetching will occur opportunistically during idle time and rotational latency periods[1]. Sometimes, however, the firmware will decide that a sequential read pattern will be better served by delaying foreground requests for further prefetching. An external scheduler is unlikely to know the exact algorithms used for replacement, prefetching, or write-back (if used). As a result, cache hits and prefetch activities will often surprise it.

In-drive scheduling. Modern disks support command queueing, and they internally schedule queued requests to maximize efficiency. An external scheduler that wishes to maintain control must either avoid command queueing or anticipate possible modification of its decisions.

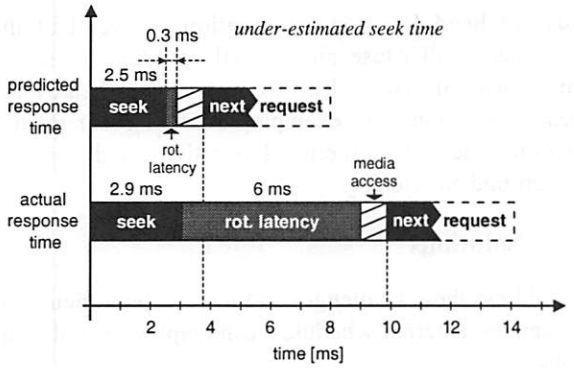
Computation of rotational offsets. A disk's rotation speed may vary slightly over time. As a result, an external scheduler must occasionally resynchronize its understanding of the disk's rotational offset. Also, whenever making a scheduling decision, it must update its view of the current offset.

Internal disk activities. Disk firmware must sometimes execute internal functions (e.g., thermal recalibration) that are independent of any external requests. Unless a

[1] Freeblock scheduling often removes the disk's opportunity to prefetch during rotational latency periods. It does so to fetch known-to-be-wanted data, which we argue is a more valuable activity. In part, we assert this because the lost prefetching will rarely eliminate subsequent media accesses, since the prefetched sectors are usually not forward in LBN order and not aligned to any block boundary or size.



(a) **Seek time over-estimation.** The larger predicted seek of 3.3 ms suggests a full rotation, resulting in a predicted response time of 10.2 ms. Since the actual seek is smaller (3.0 ms), the extra rotation does not occur and the request completes in 4.2 ms, resulting in a −6.0 ms prediction error.



(b) **Seek time under-estimation** The predicted seek of 2.5 ms results in a prediction of rotational latency of 0.3 ms and a predicted response time of 3.8 ms. Since the actual seek is larger (2.9 ms), the disk will suffer an extra rotation resulting in a response time of 9.8 ms. The prediction error is +6.0 ms.

Figure 3: **The effects of mispredicted seek times.**

device driver uses recent S.M.A.R.T. interface extensions to avoid these functions, an unexpected internal activity will occasionally invalidate the scheduler's predictions.

3.2 Consequences

The challenges listed above have five main consequences on the operation of an external fine-grained disk scheduler.

Complexity. Both the initial configuration and runtime operation of an external scheduler will be complex and disk-specific. As a result, substantial engineering may be required to achieve robust, effective operation. Worse, effective freeblock scheduling requires very accurate service time predictions to avoid disrupting foreground request performance.

Seek misprediction. When making a scheduling decision, the scheduler predicts the mechanical delays that will be incurred for each request. When there are small errors in the initial configuration of the scheduler or variations in seek times for a given cylinder distance, the scheduler will sometimes mispredict the seek time. When it does, it will also mispredict the rotational latency.

When a scheduler over-estimates a request's seek time (see Figure 3(a)), it may incorrectly decide that the disk head will "just miss" the desired sectors and have to wait almost a full rotation. With such a large predicted delay, the scheduler is unlikely to select this request even though it may actually be the best option.

When the scheduler under-estimates a request's seek time (see Figure 3(b)), it may incorrectly decide that the disk head will arrive just in time to access the desired sectors with almost no rotational latency. Because of the small predicted delay, the scheduler is likely to select this request even though it is probably a bad choice.

Under-estimated seeks can cause substantial unwanted extra rotations for foreground requests. Over-estimated seeks usually do not cause significant problems for foreground scheduling, because selecting the second-best request usually results in only a small penalty. When the foreground scheduler is used in conjunction with a freeblock scheduler, however, an over-estimated seek may cause a freeblock request to be inserted in place of an incorrectly predicted large rotational latency. Like a self-fulfilling prophecy, this will cause an extra rotation before servicing the next foreground request even though it would not otherwise be necessary.

Idle disk head time. The response time for a single request includes mechanical actions, bus transfers, and command processing. As a result, the read/write head can be idle part of the time, even while a request is being serviced. Such idleness occurs most frequently when acquiring and utilizing the bus to transfer data or completion messages. Although an external scheduler can be made to understand such inefficiencies, they can reduce its ability to utilize the potential free bandwidth found in foreground rotational latencies.

Incorrectly-triggered prefetching. Freeblock scheduling works best when it picks up blocks on the source or destination tracks of a foreground seek. However, if the disk observes two sequential READs, it may assume a sequential access pattern and initiate prefetching that causes a delay in handling subsequent requests. If one of these READs is from the freeblock scheduler, the disk will be acting on misinformation since the foreground workload may not be sequential.

Loss of head location information. Several of the challenges will cause an external scheduler to sometimes make decisions based on inaccurate head location information. For example, this will occur for unexpected cache hits, internal disk activity, and triggered foreground prefetching.

3.3 Solutions

To address these challenges and to cope with their consequences, external schedulers can employ several solutions.

Automatic disk characterization. An external scheduler must have a detailed understanding of the specific disk for which it is scheduling requests. The only practical option is to have algorithms for automatically discovering the necessary configuration information, including LBN-to-physical mappings, seek timings, rotation speed, and command processing overheads. Fortunately, mechanisms [30] and tools [23] have been developed for exactly this purpose.

Seek conservatism. To address seek time variance and other causes of prediction errors, an external scheduler can add a small "fudge factor" to its seek time estimates. By conservatively over-estimating seek times, the external scheduler can avoid the full rotation penalty associated with under-estimation. To maximize efficiency, the fudge factor must balance the benefit of avoiding full rotations with the lost opportunities inherent to over-estimation. For freeblock scheduling decisions, a more conservative (i.e., higher) fudge factor should be selected to prefer less-utilized free bandwidth opportunities to extra full rotations suffered by foreground requests.

Resync after each request. The continuous rotation of disk platters helps to minimize the propagation of prediction errors. Specifically, when an unexpected cache hit or internal disk activity causes the external scheduler to make a misinformed decision, only one request is affected. The subsequent request's positioning delays will begin at the same rotational offset (i.e., the previous request's last sector), independent of how many unexpected rotations that the previous request incurred.

Limited command queueing. Properly utilized, command queueing at the disk can be used to increase the accuracy of external scheduler predictions. Keeping two requests at the disk, instead of just one, avoids idling of the disk head. Specifically, while one request is transferring data over the bus, the other can be using the disk head.

In addition to improving efficiency, the overlapping of bus transfer with mechanical positioning simplifies the task of the external scheduler, allowing it to focus on media access delays as though the bus and processing overheads were not present. When the media access delays dominate, these other overheads will always be overlapped with another request's media access (see Figure 4).

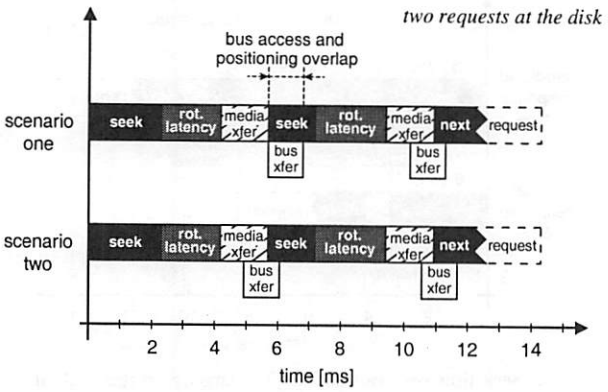


Figure 4: **Limited command queueing.** This figure repeats the two scenarios from Figure 2 but with two requests outstanding at the drive. That is, the scheduler keeps two requests at the disk — in this example, request *A* is being serviced while request *B* is queued. The drive completely overlaps the bus transfer of request *A* with the seek of request *B*, eliminating head idle time. Also, notice that the rotational latency is the same in both scenarios, making predictions easier for foreground and freeblock schedulers.

The danger with using command queueing is that the firmware's scheduling decisions may override those of the external scheduler. This danger can be avoided by allowing only two requests outstanding at a time, one in service and one in the queue to be serviced next.

Request merging. When scheduling a freeblock access to the same track as a foreground request, the two requests should be merged if possible (i.e., they are sequential and are of the same type). Not only will this merging avoid the misinformed prefetch consequence discussed above, but it will also reduce command processing overheads.

Appending a freeblock access to the end of the previous foreground request can hurt the foreground request since completion will not be reported until both requests are done. This performance penalty is avoided if the freeblock access is prepended to the beginning of the next foreground request.

4 Implementation

This section describes our implementation of an external freeblock scheduler and its integration into the FreeBSD 4.0 kernel.

4.1 Architecture

Figure 5 illustrates our freeblock scheduler's architecture, which consists of three major parts: a foreground scheduler, a freeblock scheduler, and a common dispatch queue that holds requests selected by the two schedulers.

The foreground scheduler keeps up to two requests in the dispatch queue; the remaining pending foreground requests are kept in a pool. When a foreground request completes, it is removed from the dispatch queue, and a new request is selected from the pool according to the foreground scheduling policy. This newly-selected request is put at the end of the dispatch queue. Such just-in-time scheduling allows the scheduler to consider recent requests when making decisions.

The freeblock scheduler keeps a separate pool of pending freeblock requests. When invoked, it inspects the dispatch queue and, if there is a foreground request waiting to be issued to the disk, it identifies a suitable freeblock candidate from its pool. The identified freeblock request is inserted ahead of the foreground request. The freeblock scheduler will continue to refine its choice in the background, if there is available CPU time. The device driver may send the current best freeblock request to the disk at any time. When it does so, it sets a flag to tell the freeblock scheduler to end its search.

Whenever there are fewer than two requests at the disk, the device driver issues the next request in the dispatch queue. By keeping two requests at the disk, the driver achieves the desired overlapping of bus and media activities. By keeping no more than two, it avoids reordering within the disk firmware; at any time, one request may be in service and the other waiting at the disk.

The diagram in Figure 5 shows a situation when there are two outstanding requests at the disk: a freeblock request `fb1` is currently being serviced and a foreground request `fore1` is queued at the disk. When the disk completes the freeblock request `fb1`, it immediately starts to work on the already queued request `fore1`. When the device driver receives the completion message for `fb1`, it issues the next request, labeled `fb2`, to the disk. It also sets the "stop" flag to inform the freeblock scheduler. When the foreground request `fore1` completes, the device driver sends `fore2` to the disk, tells the foreground scheduler to select a new foreground request, and (if appropriate) invokes the freeblock scheduler.

4.2 Foreground scheduler

Our foreground scheduler implements three scheduling algorithms: SSTF, SPTF, and SPTF-SWn%. SSTF is representative of the seek-reducing algorithms used by many external schedulers. SPTF yields lower foreground service times and lower rotational latencies than SSTF;

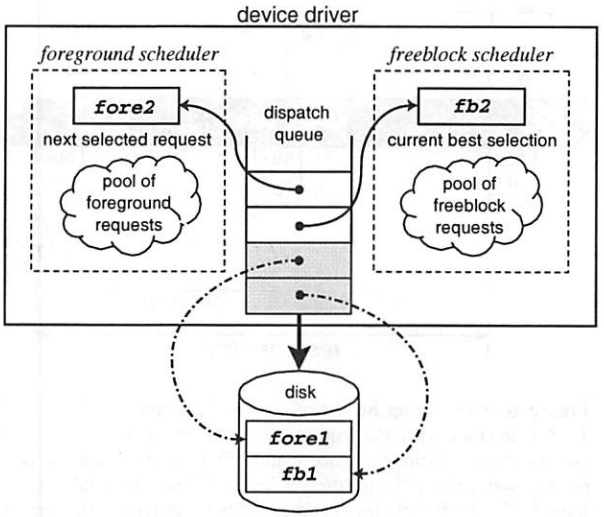


Figure 5: **Freeblock scheduling inside a device driver.**

SPTF requires the same detailed disk knowledge needed for freeblock scheduling. SPTF-SWn% was proposed to select requests with both small total positioning delays and large rotational latency components [14]. It selects the request with the smallest seek time component among the pending requests whose positioning times are within n% of the shortest positioning time.

Request timing predictions. For the SPTF and SPTF-SWn% algorithms, the foreground scheduler predicts request timings given the current head position. Specifically, it predicts the amount of time that the disk head will be dedicated to the given request; we call this time *head time*. When using command queueing, the bus activity is overlapped with positioning and media access, reducing the head time to seek time, rotational latency, and media transfer. Figure 6 illustrates the head time components that must be accurately predicted by the disk model.

The disk model in our implementation is completely parametrized; that is, there is no hard-coded information specific to a particular disk drive. The parameters fall into three categories: complete layout information with slipping and defects, seek profile, and head switch time. All of these parameters are extracted automatically from the disk using the DIXtrac tool [23]. The seek profile is used for predicting seek times, and the layout information and head switch time are used for predicting rotational latencies and media transfer times.

The layout information is a compact representation of all LBN mappings to the physical sector locations (described by a sector-head-cylinder tuple). It includes information about defects and their handling via slipping or remapping to spare sectors. It also includes skews between two successive LBNs mapped across a track,

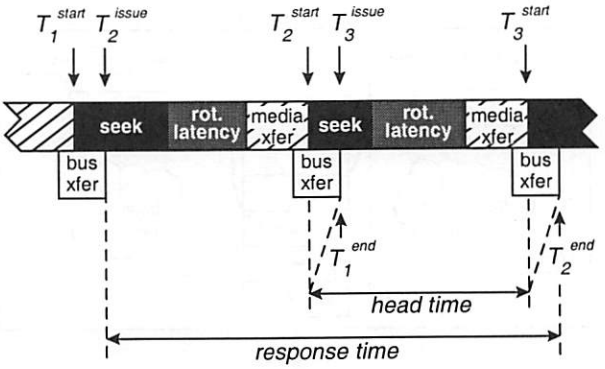


Figure 6: **Computing head time.** The head time is $T_2^{end} - T_1^{end}$. T^{issue} is the time when the request is issued to the disk, T^{start} is when the disk starts servicing the request, and T^{end} is when completion is reported. Notice that T^{issue} is different from T^{start} and that total response time, $T_2^{end} - T_2^{issue}$ includes (a portion) of bus transfer and the time the request is queued at the disk.

cylinder, or zone boundary. To achieve the desired prediction accuracy, the skews are recorded as a fraction of a revolution—using just an integral number of sectors does not give the required resolution.

The seek profile is a lookup table that gives the expected seek time for a given distance in cylinders. The table includes more values for shorter seek distances (every distance between cylinder 1–10, cylinders, every 2^{nd} for 10–20, every 5^{th} for 20–50, every 10^{th} for 50–100, every 25^{th} for 100–500, and every 100^{th} for distance beyond 500). Values not explicitly listed in the table are interpolated. Since the listed seek times are averages of seeks of a given distance, a specific seek time may differ by tens of μs depending on the distance and the conditions of the drive. Thus, the scheduler may include an explicit conservatism value to account for this variability.

4.3 Freeblock scheduler

The freeblock scheduler computes the rotational latency for the next foreground request, and determines which pending freeblock request could be handled in this opportunity. Determining the latter involves computing the extra seek time involved in going to the candidate's location and determining whether all of the necessary blocks could be fetched in time to seek to the location of the foreground request without causing a rotational miss.

The current implementation of our freeblock scheduling algorithm focuses on the goal of scanning the entire disk by touching each block of the disk exactly once. Therefore, it keeps a bitmap of all blocks with the already-touched blocks marked. When a suitable set of blocks is selected from the bitmap, the freeblock scheduler creates a disk request to read them.

The scheduling algorithm greedily tries to maximize the number of blocks read in each opportunity. To reduce search time, it searches the bitmap, looking for the most promising candidates. It starts by considering the source and destination tracks (the locations of the current and next foreground requests) and then proceeds to scan the tracks closest to the two tracks. It keeps scanning progressively farther and farther away from the source and destination tracks until it is notified via the stop flag or reaches the end of the disk. If a better free bandwidth opportunity is found, the scheduler creates a new request that replaces the previous best selection.

In early experimentation, we found that two requests on the same track often trigger aggressive disk prefetching. When the foreground workload involves sequentiality, this can be highly beneficial. Unfortunately, a freeblock request to the same track can make a random foreground workload appear to have some locality. In such cases, the disk firmware may incorrectly assume that aggressive prefetching would improve performance.

To avoid such incorrect assumptions, our freeblock scheduling algorithm will not issue a separate request on the same track. To reclaim some of the flexibility lost to this rule, it will coalesce same-track freeblock fetches with the next foreground request. That is, it will lower the starting LBN and increase the request size when blocks on the destination track represent the best selection. When the merged request completes, the data are split appropriately.

Request merging only works when the selected freeblock request is on the same (destination) track as the next foreground request. Recall that the in-service foreground request cannot be modified, since it is already queued at the disk. For this reason, our freeblock scheduler will not consider a request that would be on the source track.

Avoiding incorrect triggering of the prefetcher also prevents another same-track case: any freeblock opportunity that spans contiguous physical sectors that hold non-contiguous ranges of LBNs (i.e., they cross the logical beginning of the track). To read all of the sectors would require two distinct requests, because of the LBN-based interface. However, since these two freeblock requests might trigger the prefetcher, the algorithm considers only the larger of the two.

4.4 Kernel implementation

We have integrated our scheduler into the FreeBSD 4.0 kernel. For SCSI disks (`/dev/da`), the foreground scheduler replaces the default C-LOOK scheduler implemented by the `bufqdisksort()` function. Just like the default C-LOOK scheduler, our foreground scheduler is called from the `dastart()` function and it puts

requests onto the device's queue, *buf_queue*, which is the dispatch queue in Figure 5. This queue is emptied by `xpt_schedule()`, which is called from `dastart()` immediately after the call to the scheduler.

The only architectural modification to the direct access device driver is in the return path of a request. Normally, when a request finishes at the disk, the `dadone()` function is called. We have inserted into this function a callback to the foreground scheduler. If the foreground scheduler selects another request, it calls `xpt_schedule()` to keep two requests at the disk. When the callback completes, `dadone()` proceeds normally.

The freeblock scheduler is implemented as a kernel thread and it communicates with the foreground scheduler via a few shared variables. These variables include the *restart* and *stop* flags and the pointer to the next foreground request for which a freeblock request should be selected.

Before using the freeblock scheduler on a new disk, the disk performance attributes for the disk model must first be obtained by the DIXtrac tool [23]. This one time cost of 3–5 minutes can be a part of an augmented *newfs* process that stores the attributes along with the superblock and inode information.

The current implementation generates freeblock requests for a disk scan application from within the kernel. The full disk scan starts when the disk is first mounted. The data received from the freeblock requests do not propagate to the user level.

4.5 User-level implementation

The scheduler can also run as a user-level application. In fact, the FreeBSD kernel implementation was originally developed as a user-level application under Linux 2.4. The user-level implementation bypasses the buffer cache, the file system, and the device driver by assembling SCSI commands and passing them directly to the disk via Linux's SCSI generic interface.

In addition to easier development, the user-level implementation also offers greater flexibility and control over the location, size, and issue time of foreground requests during experiments. For the in-kernel implementation, the locations and sizes of foreground accesses are dictated by the file system block size and read-ahead algorithms. Furthermore, the file system cache satisfies many requests with no disk I/O. To eliminate such variables from the evaluation of the scheduler effectiveness, we use the user-level setup for most of our experiments.

	Quantum Atlas 10k	Seagate Cheetah 18LP
Year	1999	1998
RPM	10000	10016
Head switch (ms)	0.8	1.0
Avg. seek (ms)	5.0	5.4
Number of heads	6	6
Sectors per track	334–224	360–230
Bandwidth (MB/s)	27–18	28–18
Capacity (GB)	9	9
Zero-latency access	yes	no

Table 1: **Disk characteristics.**

5 Evaluation

This section evaluates the external freeblock scheduler, showing that its service time predictions are very accurate and that it is therefore able to extract substantial free bandwidth. As expected, it does not achieve the full performance that we believe could be achieved from within disk firmware — it achieves approximately 65% of the predicted free bandwidth. The limitations are explained and quantified.

5.1 Experimental setup

Except where otherwise specified, our experiments are run on the Linux version of the scheduler. The system hardware includes a 550MHz Pentium III, 128 MB of main memory, an Intel 440BX chipset with a 33MHz, 32bit PCI bus, and an Adaptec AHA-2940 Ultra2Wide SCSI controller. The experiments use 9GB Quantum Atlas 10k and Seagate Cheetah 18LP disk drives, whose characteristics are listed in Table 1. The system is running Linux 2.4.2. The experiments with the FreeBSD kernel implementation use the same hardware.

Unless otherwise specified, the experiments use a synthetic foreground workload that approximates observed OLTP workload characteristics. This synthetic workload models a closed system with per-task disk requests separated by think times of 30 milliseconds. The experiments use a multiprogramming level of ten, meaning that there are ten requests active in the system at any given point. The OLTP requests are uniformly-distributed across the disk's capacity with a read-to-write ratio of 2:1 and a request size that is a multiple of 4 KB chosen from an exponential distribution with a mean of 8 KB. Validation experiments (in [21]) show that this workload is sufficiently similar to disk traces of Microsoft's SQL server running TPC-C for the overall freeblock-related insights to apply to more realistic OLTP environments.

The background workload consists of a single freeblock

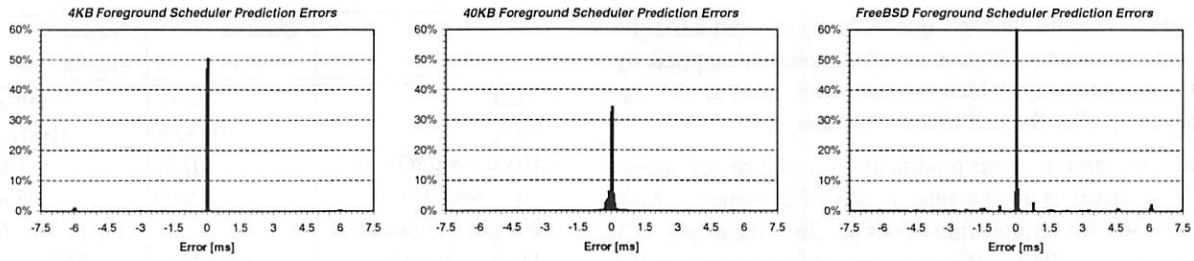


Figure 7: **PDFs of prediction error for foreground requests on a Quantum Atlas 10k disk.** The three graphs show the distribution of differences between the scheduler's predicted head time and the observed time. Negative values denote over-estimation, which means that the scheduler predicted a longer service time than was measured. The first graph shows the distribution of prediction errors for the user-level foreground workload with 4KB average request size. The second graph shows the distribution of prediction errors for the user-level foreground workload with 40KB average request size. The third graph shows the distribution of prediction errors for the FreeBSD system running the random small file read workload.

read request for the entire capacity of the disk. That is, the freeblock scheduler is asked to fetch each disk sector once, but with no particular order specified.

5.2 Service time prediction accuracy

Central to all fine-grain scheduling algorithms is the ability to accurately predict service times. Figure 7 shows PDFs of error in the external scheduler's head time predictions for the Atlas 10k disk. For random 4 KB requests, 97.5% of requests complete within 50 μs of the scheduler's prediction. The other 1.8% of requests take one rotation longer than predicted, because the seek time was slightly underpredicted and the remaining 0.7% took one rotation shorter than predicted. For the Cheetah 18LP disk, 99.3% of requests complete within 50 μs of the scheduler's prediction and the other 0.7% take one rotation longer or shorter than predicted. We have verified that more localized requests (e.g., random requests within a 50 cylinder range) are predicted equally well.

For random 40 KB requests to the Atlas 10k disk, 75% of requests complete within 150 μs of the scheduler's predictions. The disk head times for larger requests are predicted less accurately mainly because of variation in the overlap of media transfer and bus transfer. For example, one request may overlap by 100 μs more than expected, which will cause the request completion to occur 100 μs earlier than expected. In turn, because the next request's head time is computed relative to the previous request's end time, this extra overlap will usually cause the next request prediction to be 100 μs too low. (Recall that media transfers always end at the same rotational offset, normalizing such errors.) But, because the prediction errors are due to variance in bus-related delays rather than media access delays, they do not effect the external scheduler's effectiveness; this fact is particularly important for freeblock scheduling, which explicitly tries to create large background transfers.

The FreeBSD graph in Figure 7(c) shows the prediction error distribution for a workload of 10,000 reads of randomly chosen 3 KB files. For this workload, the file system was formatted with a 4 KB block size and populated with 2000 directories each holding 50 files. Even though a file is chosen randomly, the file system access pattern is not purely random. Because of FFS's access to metadata that is in the same cylinder group as the file, some accesses are physically localized or even to the same track, which can trigger disk prefetching.

For this workload, 76% of all requests were correctly predicted within 150 μs. 5% of requests, at $\pm$800 μs, are due to bus and media overlap mispredictions. There are 4% of +6 ms mispredictions that account for an extra full rotation. An additional 4% of requests at -7.5 ms misprediction were disk cache hits. Finally, 8% of the requests are centered around $\pm$1.5 and $\pm$4.5 ms. These requests immediately follow surprise cache hits or unexpected extra rotations and are therefore mispredicted.

To objectively validate the external scheduler, Figure 8 compares the three external algorithms (SSTF, SPTF, and SPTF-SW60%) with the disk's in-firmware scheduler. As expected, SPTF outperforms SPTF-SW60% which outperforms SSTF, and the differences increase with larger queue depths. The external scheduler's SPTF matches the Atlas 10k's ORCA scheduler [20] (apparently an SPTF algorithm), indicating that their decisions are consistent. We observed the same consistency between the external scheduler's SPTF and the Cheetah 18LP's in-firmware scheduler.

5.3 Freeblock scheduling effectiveness

To evaluate the effectiveness of our external freeblock scheduler, we measure both foreground performance and achieved free bandwidth. We hope to see significant free bandwidth achieved and no effect on foreground performance.

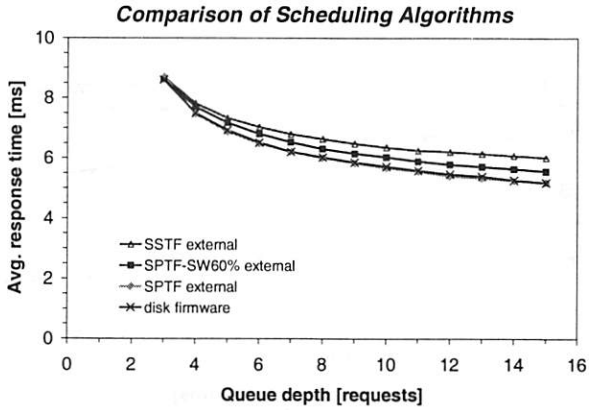


Figure 8: **Measured performance of foreground scheduling algorithms on a Quantum Atlas 10k disk.** The top three lines represent the external scheduler using SSTF, SPTF-SW60% and SPTF. The fourth line shows performance when all requests are given immediately to the Quantum Atlas 10k, which uses its internal scheduling algorithm. The "disk firmware" line exactly overlaps the "SPTF external" line, simultaneously indicating that the firmware uses SPTF and that the external scheduler makes good decisions. Linux's default limit on requests queued at the disk is 15 (plus one in service).

How well it works. Figure 9 shows both performance metrics as a function of the freeblock scheduler's seek conservatism. This conservatism value is only added to the freeblock scheduler's seek time predictions, reducing the probability that it will under-estimate a seek time and cause a full rotation. As conservatism increases, foreground performance approaches its no-freeblock-scheduling value. Foreground performance is reduced by <2% at 0.3 ms of conservatism and by <0.6% at 0.4 ms. The corresponding penalties to achieved free bandwidth are 3% and 10%.

All three foreground scheduling algorithms are shown in Figure 9. As expected, the highest foreground performance and the lowest free bandwidth are achieved with SPTF. SSTF's foreground performance is 13–15% lower, but it provides for 2.1–2.6× more free bandwidth. SPTF-SW60% achieves over 80% of SSTF's free bandwidth with only a 5–6% penalty in foreground performance relative to SPTF, offering a nice option if one is willing to give up small amounts of foreground performance.

Limitations of external scheduling. Having confirmed that external freeblock scheduling is possible, we now address the question of how much of the potential is lost. Figure 10 compares the free bandwidth achieved by our external scheduler with the corresponding simulation results [14], which remain our optimistic expectation for in-firmware freeblock scheduling. The results show that there is a substantial penalty (≈35%) for external scheduling.

The penalty comes from two sources, with each responsible for about half. The first source is conservatism; its direct effect can be seen in the steady decline of the simulation line. The second source is our external scheduler's inability to safely issue distinct commands to the same track. When we allow it to do so, we observe unexpected extra rotations caused by firmware prefetch algorithms that are activated. We have verified that, beyond conservatism of 0.3 ms, the vertical difference between the two lines is almost entirely the result of this limitation; with the same one-request-per-track limitation, the simulation line is within 2–3% of the measured free bandwidth beyond 0.3 ms of conservatism.

Disallowing distinct freeblock requests on the source or destination tracks creates two limitations. First, it prevents the scheduler from using free bandwidth on the source track, since the previous foreground request is always previously sent to the disk and cannot subsequently be modified. (Recall that request merging allows free bandwidth to be used on the destination track without confusing the disk prefetch algorithms.) Second, and more problematic, it prevents the scheduler from using free bandwidth for blocks on both sides of a track's end. Figure 11 shows a free bandwidth opportunity than spans LBNs 1326–1334 at the end of a track and LBNs 1112–1145 at the beginning of the same track. To pickup the entire range, the scheduler would need to send one request for 9 sectors starting at LBN 1326 and a second request for 34 sectors at LBN 1112. The one-request restriction allows only one of the two. In this example, the smaller range is left unused.

5.4 CPU overhead

To quantify the CPU overhead of freeblock scheduling, we measured the CPU load on FreeBSD for the random small file read workload under three conditions. First, we established a base-line for CPU utilization by running unmodified FreeBSD with its default C-LOOK scheduler. Second, we measured the CPU utilization when running our foreground scheduler only. Third, we measured the CPU utilization when running both the foreground and freeblock schedulers.

The CPU utilization for unmodified FreeBSD was 5.1% and 5.4% for our foreground scheduler. Therefore, with negligible CPU overhead (of 0.3%), we are able to run an SPTF scheduler. The average utilization of the system running both the foreground and the freeblock schedulers was 14.1%. Subtracting the base line CPU utilization of 5.1% when running the workload gives 9% overhead for freeblock scheduling. In future work, we expect algorithm refinements to reduce this CPU overhead substantially.

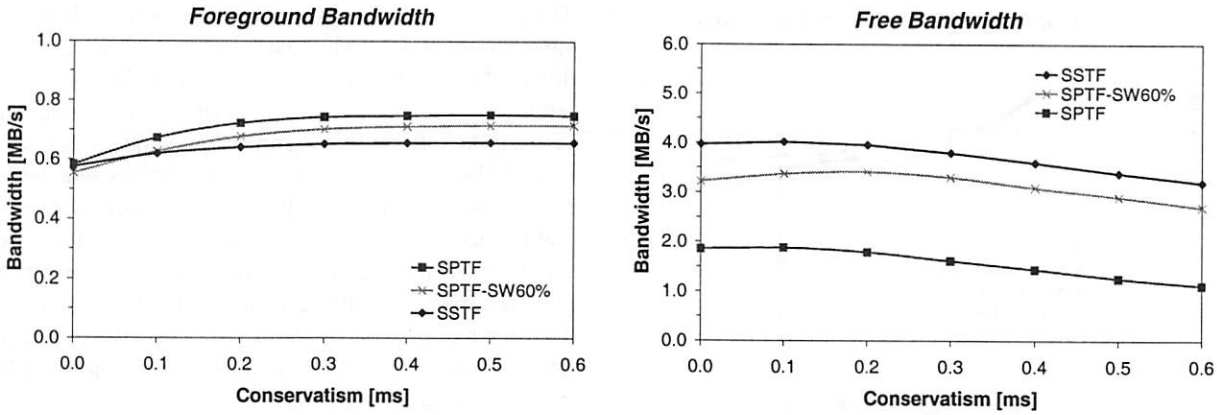


Figure 9: **Foreground and free bandwidth for a Quantum Atlas 10k as a function of seek conservatism.** The conservatism is only for freeblock scheduling decisions, which must strive to avoid overly-aggressive predictions that penalize the foreground workload. At 0.3 ms, foreground performance is 1–2% lower. At 0.4 ms, foreground performance is 0.2–0.6% lower. Note that ensuring minimal foreground impact does come at a cost in achieved free bandwidth.

Comparing the foreground and free bandwidths for the SPTF-SW60% scheduler in Figure 9 for a conservatism of 0.4 ms, the modest cost of 8% of the CPU is justified by a 6× increase in disk bandwidth utilization.

6 Related Work

Before the standardization of abstract disk interfaces, like SCSI and IDE, fine-grained request scheduling was done outside of disk drives. Since then, most external schedulers have used less-detailed seek-reducing algorithms, such as C-LOOK and Shortest-Seek-First. Even these are only approximated by treating LBNs as cylinder numbers [30].

Several research groups [1, 3, 5, 6, 26, 28, 31] have developed software-only external schedulers that support fine-grained algorithms, such as Shortest-Positioning-Time-First. Our foreground scheduler borrows its structure, its rotational position detection approach, and its use of conservatism from these previous systems. Our original pessimism regarding the feasibility of freeblock scheduling outside the disk also came from these projects—their reported experiences suggested conservatism values that were too large to allow effective freeblock scheduling. Also, some only functioned well on old disks, for large requests, or with the on-disk cache disabled. We have found that effective external freeblock scheduling requires the additional refinements described in Section 3, particularly the careful use of command queueing and the merging of same-track requests.

This paper and its related work section focus mainly on the challenge of implementing freeblock scheduling outside the disk. Lumb et al. [14] discuss work related to freeblock scheduling itself.

7 Summary

Refuting our original pessimism, this paper demonstrates that it is possible to build an external freeblock scheduler. From outside the disk, our scheduler can replace many rotational latency delays with useful background media transfers; further, it does this with almost no increase (less than 2%) in foreground service times. Achieving this goal required greater accuracy than could be achieved with previous external SPTF schedulers, which our scheduler achieves by exploiting the disk's command queueing features. For background disk scans, over 3.1 MB/s of free bandwidth (15% of the disk's total media bandwidth) is delivered, which is 65% of the simulation predictions from previous work.

Given previous pessimism that external freeblock scheduling was not possible, achieving 65% of the potential is a major step. However, our results also indicate that there is still value in exploring in-firmware freeblock scheduling.

Acknowledgements

We thank Peter Honeyman (our shepherd), John Wilkes, the other members of the Parallel Data Lab, and the anonymous reviewers for helping us refine this paper. We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support. This work is partially funded by an IBM Faculty Partnership Award and by the National Science Foundation.

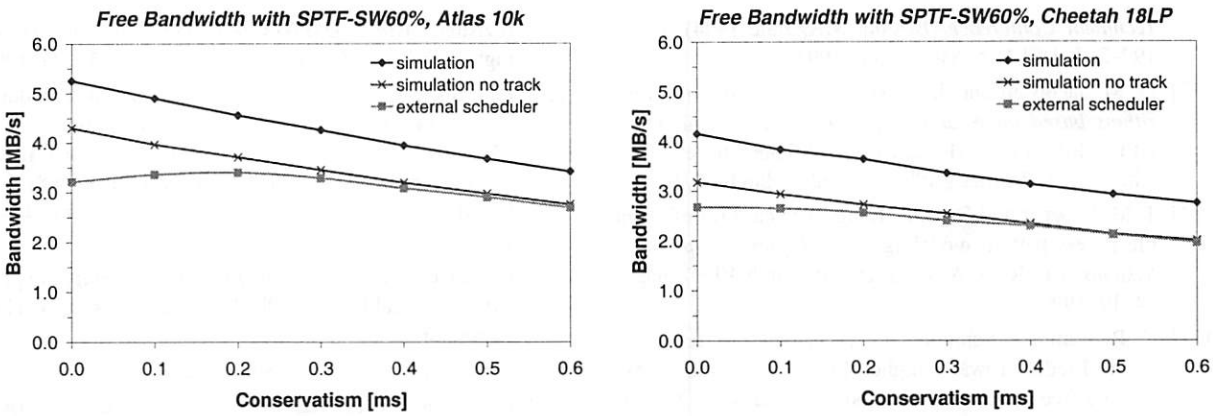


Figure 10: **Measured and simulated free bandwidth as a function of conservatism.** The line labeled simulation shows the expected free bandwidth obtained from our simulated, in-firmware freeblock scheduler operating at the given level of conservatism. The line labeled simulation no track shows a case when the simulated freeblock scheduler does not put a non-merged freeblock request on the same track as a foreground request, mimicking a major limitation of our external scheduler. The line labeled external scheduler shows the actual measured free bandwidth obtained from a disk by our freeblock scheduler implementation.

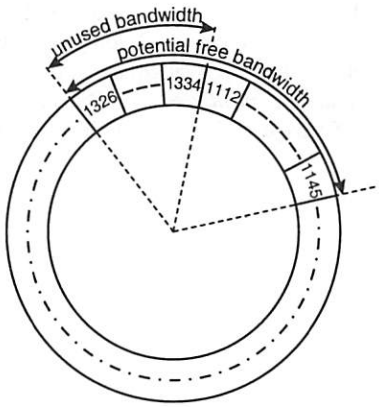


Figure 11: **A limitation of the external scheduler.** This diagram illustrates a case where the potential free bandwidth spans the start/end of a track. In this case, no single contiguous LBN range covers the potential free bandwidth. Two requests would be needed, one to LBN 1326 and one to LBN 1112. Since our scheduler can only send one free bandwidth request per track, the system will select the range from LBNs 1112-1145. This wastes the opportunity to access LBNs 1326-1334.

References

[1] M. Aboutabl, A. Agrawala, and J.-D. Decotignie. Temporally determinate disk access: an experimental approach. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI, 22–26 June 1998). Published as *Performance Evaluation Review*, **26**(1):280–281. ACM, 1998.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):10–22, 1992.

[3] P. Barham. A fresh approach to file system quality of service. *International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, MO, 19–21 May 1997), pages 113–122. IEEE, 1997.

[4] P. Biswas, K. K. Ramakrishnan, and D. Towsley. Trace driven analysis of write caching policies for disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, 1993.

[5] P. Bosch and S. J. Mullender. Real-time disk scheduling in a mixed-media file system. *Real-Time Technology and Applications Symposium* (Washington D.C., USA, 31 May – 02 June 2000), pages 23–32. IEEE, 2000.

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. *IEEE International Conference on Multimedia Computing and Systems* (Florence, Italy, 07–11 June 1999), pages 400–405. IEEE, 1999.

[7] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, **14**(4):311–343, November 1996.

[8] S. C. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, **18**(1):44–54, January 1992.

[9] P. J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18–20 April 1967), pages 9–21, April 1967.

[10] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 201–212. USENIX Association, 1995.

[11] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. *Summer USENIX*

Technical Conference (Boston, MA, June 1994), pages 197–207. USENIX Association, 1994.

[12] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position.* Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[13] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. *Hot Topics in Operating Systems* (Rio Rico, Arizona, 29–30 March 1999), pages 14–19, 1999.

[14] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 87–102. USENIX Association, 2000.

[15] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[17] A. G. Merten. *Some quantitative techniques for file organization.* PhD thesis. University of Wisconsin, Computing Centre, June 1970.

[18] S. W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[20] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB SCSI product manual*, Document number 81-119313-05, August 1999.

[21] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD International Conference on Management of Data* (Dallas, TX, 14–19 May 2000), pages 13–21, 2000.

[22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[23] J. Schindler and G. R. Ganger. *Automated disk drive characterization.* Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[24] P. H. Seaman, R. A. Lind, and T. L. Wilson. On teleprocessing system design, part IV: an analysis of auxiliary-storage activity. *IBM Systems Journal*, **5**(3):158–170, 1966.

[25] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.

[26] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Madison, WI, June 1998). Published as *Performance Evaluation Review*, **26**(1):44–55, 1998.

[27] L. Shriver. *A formalization of the attribute mapping problem.* Technical Report HPL–1999–127. Hewlett–Packard Laboratories, 1999.

[28] Trail. http://www.ecsl.cs.sunysb.edu/trail.html.

[29] R. Y. Wang, T. E. Anderson, and M. D. Dahlin. *Experience with a distributed file system implementation.* Technical Report CSD–98–986. University of California at Berkeley, January 1998.

[30] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), 1994.

[31] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.

Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload

Chi Zhang* Xiang Yu* Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

Transaction processing applications such as those exemplified by the TPC-C benchmark are among the most demanding I/O applications for conventional storage systems. Two complementary techniques exist to improve the performance of these systems. *Eager-writing* allows the free block that is closest to a disk head to be selected for servicing a write request, and *mirroring* allows the closest replica to be selected for servicing a read request. Applied individually, the effectiveness of each of these techniques is limited. An *eager-writing disk array* (EW-Array) combines these two complementary techniques. In such a system, eager-writing enables low-cost replica propagation so that the system can provide excellent performance for both read and write operations while maintaining a high degree of reliability. To fully realize the potential of an EW-Array, we must answer at least two key questions. First, since both eager-writing and mirroring rely on extra capacity to deliver performance improvements, how do we satisfy competing resource demands given a fixed amount of total disk space? Second, since eager-writing allows data to be dynamically located, how do we exploit this high degree of location independence in an intelligent disk scheduler? In this paper, we address these two key questions and compare the resulting EW-Array prototype performance against that of conventional approaches. The experimental results demonstrate that the eager-writing disk array is an effective approach to providing scalable performance for an important class of transaction processing applications.

1 Introduction

Transaction processing applications such as those exemplified by TPC-C [27] tend to pose more difficult challenges to storage systems than office workloads. These applications exhibit little locality or sequentiality; a large percentage of the I/O requests are writes, many of which are synchronous; and there may be little idle time.

Traditional techniques that work well for office workloads tend to be less effective for these transaction processing applications. Memory caching provides little relief in the presence of poor locality and a small read-to-write ratio. As disk areal density improves at 60-100% annually [9], as memory density improves at only 40% per year, and as the amount of data in transaction processing systems continues to grow, one can expect little improvement in cache hit rates in the near future. Delayed write techniques become less applicable in the presence of a large number of synchronous writes that must satisfy strict reliability requirements, requirements that are sometimes not met by even expensive NVRAM-based solutions [13]. Even when it is possible to buffer delayed writes in faster storage levels, such as an NVRAM, the poor write locality implies that there are very few overwrites before the buffered data reaches disks. Furthermore, high throughput requirements in conjunction with scarce idle time make it difficult to schedule background activities, such as de-staging from NVRAM [13, 20], garbage collection in log-structured solutions [13, 21, 22, 24], and data relocation [19], without impacting foreground I/O activities. The net effect of these challenges is that transaction processing applications tend to be more closely limited by disk latency, a performance characteristic that has seen an annual improvement of only about 10% [9].

Although the traditional caching and asynchronous I/O techniques have not been very successful, a number of other techniques have proven promising. One is mirroring: a mirrored system can improve read latency by sending a read request to the disk whose head is closest to a target replica [2, 5], and it can improve throughput by intelligently scheduling the requests in a load-balanced manner. Mirroring, unfortunately, is not without its challenges. Chief among them is the cost

*Department of Computer Science, Princeton University, {chizhang,xyu,rywang}@cs.princeton.edu.

†Department of Computer Science, Yale University, arvind@cs.yale.edu.

This work is supported in part by IBM. Wang is supported by NSF Career Award CCR-9984790. Krishnamurthy is supported by NSF Career Award CCR-9985304.

of replica propagation—each write request is turned into multiple physical writes that compete for I/O bandwidth with normal I/O operations. High update rates, the lack of idle time for masking replica propagation, and poor locality only make matters worse.

While mirroring is more effective for improving a read-dominant workload, a technique called *eager-writing* is more effective for improving a write-dominant workload. Eager-writing refers to the technique of allocating a free block that is closest to the current disk head position to satisfy a write request [4, 6, 28]. Under the right conditions, by eliminating almost all of seek and rotational delay, eager-writing can deliver very fast write performance without compromising reliability guarantees, even for workloads that comprise of synchronous I/Os and have poor locality. What eager-writing does not address, however, is read performance.

Since data replication in a mirrored system improves read performance, and since eager-writing improves write performance, reduces the cost of replica propagation, and ensures a high degree of data reliability, it is only natural to integrate these two techniques so that we may harvest the best of both worlds. We call the result of this integration an *eager-writing array* or an *EW-Array*: in the simplest form, an EW-Array is just a mirrored system that supports eager-writing.

This integration, however, is not without its own tension. In order to achieve good write performance under eager-writing, one must reserve enough disk space to ensure that an empty block can be located close to the current disk head position. At the same time, to achieve good read performance under mirroring, the system needs to devote disk space to store a sufficient number of replicas so that it can choose a conveniently located replica to read. Given a fixed budget of disks, one must resolve this tension by carefully balancing the number of disks devoted to each of these two dimensions. To further complicate the matter, striping can improve both read and write performance, so one must also consider this third dimension of the number of disks devoted to striping. Although configuring a storage system based on the number of disk heads instead of capacity for TPC-C-like workloads is a common practice, and some previous studies such as the "Doubly Distorted Mirror" have incorporated "write-anywhere" elements in a disk array [19], what is not well understood is how to balance the number of disks devoted to each one of the mirroring, eager-writing, and striping dimensions to get the most out of a given number of disks.

While properly *configuring* an EW-Array along these three dimensions presents one challenge, request *scheduling* on such a disk array presents another. In the request queue of a traditional update-in-place storage system, the locations of all the queued requests are known. Although the scheduler can sometimes choose among several mirrored replicas to satisfy a request, the degree of freedom is limited. This is no longer the case for an EW-Array: while the locations of the read requests are known, the scheduler has the freedom of choosing *any* combination of free blocks to satisfy the write requests. Although disk scheduling is a well-studied problem in conventional systems, what is not well understood is how a good scheduler can exploit this large degree of freedom to optimize throughput.

The main contributions of this paper are:

- a disk array design that integrates eager-writing with mirroring in a balanced configuration to provide the best read and write performance for a transaction processing workload,
- a number of disk array scheduling algorithms that can effectively exploit the flexibility afforded by the location-independent nature of eager writing, and
- evaluation of a number of alternative strategies that share the common goal of improving performance by introducing extra disk capacity.

We have designed and implemented a prototype EW-Array. Our experimental results demonstrate that the EW-Array can significantly outperform conventional systems. For example, under the TPC-C workload, a properly configured EW-Array delivers 1.4 to 1.6 times lower latency than that achieved on highly optimized striping and mirroring systems. The same EW-Array achieves approximately 2 times better sustainable throughput.

The remainder of this paper is organized as follows. Section 2 motivates the integration of eager-writing with mirroring in an EW-Array. Section 3 explores different EW-Array configurations as we change the way the extra disk space is distributed. Section 4 analyzes a number of new disk scheduling algorithms that exploit the location independent nature of eager-writing. Section 5 describes the integrated simulator and prototype EW-Array. The experimental results of Section 6 evaluate a wide range of disk array configuration alternatives. Section 7 describes some of the related work. Section 8 concludes.

2 Eager-Writing Disk Arrays

In this section, we explain how eager-writing, mirroring, striping, and the combination of these techniques can effectively improve the performance of TPC-C-like applications.

2.1 Eager-writing

In a traditional update-in-place storage system, the addresses of the incoming I/O requests are mapped to fixed physical locations. In contrast, under eager-writing, to satisfy a write request, the system allocates a new free block that is closest to the current disk head position [4, 6, 28]; consequently, a logical address can be mapped to different physical addresses at different times.

A number of characteristics associated with eager-writing make it suitable for transaction processing applications. The chief advantages of eager-writing are excellent small write performance (in terms of both latency and throughput) and a high degree of reliability. The main component of the eager-writing latency is the time it takes for the closest free block to rotate under the disk head. Even at a relatively high disk utilization of 80% and a disk block size of 4 KB, this latency is well below 1 m*s* and can be made even lower with lower disk utilization. Furthermore, the improvement of this latency scales with that of platter bandwidth, which is improving much more quickly than seek and rotational delays experienced by update-in-place systems. This performance advantage of eager-writing is particularly appealing to a TPC-C-like workload, which has a large percentage of small writes. By committing the data synchronously to the disk platter, eager-writing also achieves a high degree of data reliability, a degree of reliability that is unmatched by even NVRAM-based solutions, which typically have far worse mean-time-to-failure characteristics [13].

Of course, no storage system can cater to all workloads equally successfully, and eager-writing is certainly no exception. One example is frequent sequential reads following random updates—eager-writing would destroy locality during the random updates, thus resulting in poor sequential read performance. One possible remedy is periodic data reorganization that restores physical data sequentiality. Fortunately, such complications do not arise in TPC-C-like workloads, which are characterized by small reads and writes with little locality. Another difficulty that may arise with eager-writing is caused by an uneven distribution of free blocks. For example, if free blocks are concentrated in one part of the disk but the disk head is forced by read requests into regions with few free blocks, then a subsequent write may suffer a long delay. Fortunately, such complications do not arise with TPC-C-like workloads either. Indeed, the random writes of TPC-C cause the free blocks to be evenly distributed throughout the disk under eager-writing; this is desirable because a free block is never very far from the current head position.

In short, transaction processing workloads like TPC-C can benefit a great deal from the performance and reliability advantages offered by eager-writing, while the very nature of the workload allows it to avoid the performance pitfalls of eager-writing.

2.2 Mirroring and Striping

A D_m-way mirror, in addition to ensuring a high degree of reliability, can improve small read performance in terms of both latency and throughput. It can improve latency because the system can schedule the disk head that is closest to a replica to satisfy a read request [2, 5]. It can improve throughput because any request can be satisfied by any disk, and an intelligent scheduler should be able to exploit the freedom in distributing the incoming requests to balance load.

Although cost per byte and capacity per drive remain the predominant concerns of the consumer market, due to the large cost and performance gaps between disk and memory, database vendors have long recognized the need for trading capacity to obtain higher performance while configuring storage systems. A D_m-way mirror is just one of the ways to improve performance by exploiting excess capacity. This approach, however, has an obvious limitation—as one increases the degree of replication, the cost of replica propagation becomes prohibitive. One possible way of addressing this high cost is to perform some of the propagations in the background during idle periods. Unfortunately, TPC-C-like workloads are characterized by a combination of high write ratio and scarce idle time, a combination that makes it difficult to realize the potential benefits of mirroring.

An alternative to mirroring is striping—by partitioning and distributing data across a D_s-way striped system, the system reduces the maximum seek distance by a factor of D_s as only a fraction of each disk is used. This is attractive compared to mirroring because there is no replica propagation cost. Unlike mirroring, unfortunately, striping cannot reduce rotational delay. As we raise D_s, only the seek time is lowered and that too at a diminishing rate. Furthermore, unlike mirroring, due to the partitioning of data, the choice of which disk to send a request to is limited, so it is more difficult to

perform load-balancing.

In practice, disk array designers have used a combination of mirroring and striping to form a *striped mirror* [3, 11, 26]. In a $D_m \times D_s$ striped mirror, data is partitioned into D_s sets, each of which is replicated D_m times. The configuration where $D_m = 2$ is commonly referred to as "RAID-10". The replica propagation cost remains an obstacle to achieving good performance on RAID-10; and one seldom chooses a replication factor D_m that is greater than two.

2.3 Eager-writing Disk Arrays

An EW-Array resembles a conventional striped mirror in how data is distributed and reads are performed. However, the two systems differ in how writes are satisfied: instead of performing a write to one of many fixed locations, a $D_m \times D_s$ EW-Array chooses a disk whose head is closest to a free block among D_m candidates to perform the foreground write. In cases where a higher degree of reliability is desired, the two disk heads that are closest to their free blocks are chosen to perform the foreground writes. The remaining $D_m - 1$ (or $D_m - 2$) writes are buffered in the *delayed write queues* of the remaining disks to be performed in the background, also in an eager-writing fashion.

In an EW-Array, reads enjoy good latency and throughput just as they do in a conventional striped mirror. Foreground write latency is improved greatly due to eager-writing. This latency can be even lower when there are more disk heads to choose from. Unlike a striped mirror, copy propagation is no longer the limiting problem because the writes are sufficiently efficient that they are easily masked even when idle time is scarce. As a result, an EW-Array can sustain higher I/O throughput. The low cost of replica propagation also makes it possible to raise the degree of replication D_m for even lower read latency or to increase the fraction of foreground writes for higher reliability.

3 Configuring an EW-Array

An EW-Array combines three techniques: eager-writing, mirroring, and striping. One commonality shared by all three of these techniques is that they all need extra disk capacity to be effective. We first examine individually how performance under each technique improves in response to increased capacity. We then analyze their combined effect. We use simple random workloads and simulation results in this section to study these techniques. More details about the simulation environment and results from more realistic workloads will be presented in later sections.

3.1 Impact of Extra Space on Eager-writing

In order for eager-writing to be effective, one needs to reserve enough extra space so that a free block can always be located near the current disk head position. Let disk utilization be U; we define *dilution* to be $D_d = 1/U$. For example, when $D_d = 2$, we use twice as much capacity as is necessary.

Figure 1(a) shows how the components of the average write cost respond to different dilution factors (D_d) under a simple random write workload running on a 10,000 RPM Seagate disk (ST39133LWV). (In this case, $D_m = D_s = 1$. The block size is 4 KB, and there is no queueing.) In this figure (and the rest of this paper), *overhead* is defined to include various processing times and transfer costs. Under eager writing, when the closest free block is located in the current track, only rotational latency is incurred. When the closest free block is located in a neighboring track, a track switch or a small seek is also needed, and this time is counted as seek time.

As we increase the amount of extra space, both the rotational delay and seek time decrease as the disk head travels a shorter distance to locate the nearest free block. This improvement reaches diminishing return as the overhead dominates.

3.2 Impact of Extra Space on Mirroring

Figure 1(b) shows how the components of the average read cost respond to different degrees of replication (D_m) in a mirrored system under a random read workload. (In this case, $D_d = D_s = 1$.) The read overhead is lower than the write overhead, because it takes longer for the disk head to settle when servicing a write request. Note that mirroring reduces both the seek and rotational delays of read requests. These components, however, remain significant if mirroring is the only technique employed.

3.3 Impact of Extra Space on Striping

Figure 1(c) shows how the components of the average read cost respond to different degree of striping (D_s) under a random read workload. (In this case, $D_d = D_m = 1$.) By restricting the disk head within a small seek distance, striping lowers seek delay. Unlike mirroring, it has no impact on rotational delay. As D_s increases, rotational delay dominates if striping is the only technique employed.

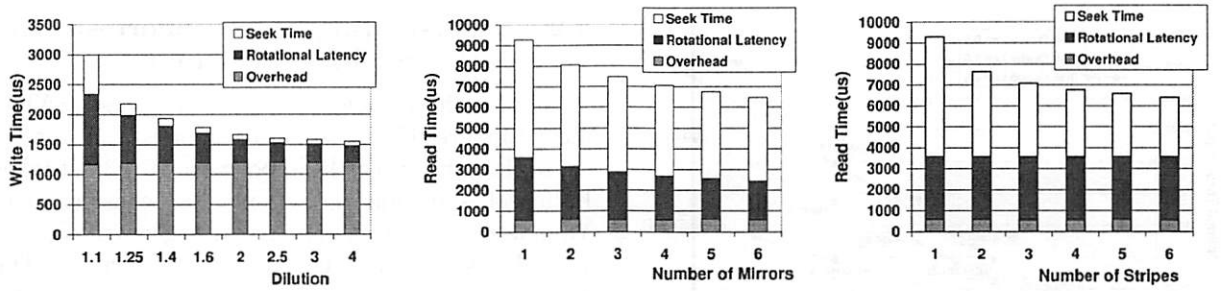


Figure 1: Components of average write response time as functions of (a) the degree of dilution D_d, (b) the degree of replication D_m, and (c) the degree of striping D_s.

3.4 Distributing Extra Space in an EW-Array

An EW-Array employs all three of the above techniques. A large D_d value allows for more efficient writes. A large D_s value more aggressively reduces the seek cost. A large D_m value more aggressively reduces the rotational cost of reads. Given a total budget of D disks and the constraint $D = D_d \times D_m \times D_s$, one must carefully balance these three dimensions to optimize the overall performance. The decision of how to configure these three dimensions is influenced by both the workload and disk characteristics. A workload that has a small read-to-write ratio and little idle time demands a large dilution factor D_d so that more resources are devoted to speeding up writes. Disks with large seek delays demand a large striping factor D_s, while disks with large rotational delay demand a large mirroring factor D_m.

In this section, we explore the impact of array configurations using a simple synthetic workload (that is part of the Intel "Iometer" benchmark [15]). More complex workloads are explored in Section 6. In each of the test runs, the length of the queue of the outstanding requests is kept at a constant. This is accomplished by adding a new request to the queue as soon as an old one is retired from it. Different queue lengths emulate different degree of idleness in the system. In all runs, the read/write ratio is 50/50.

Figure 2 compares the latency of alternative EW-Array configurations. In these experiments, the number of outstanding requests is one so there is no queueing. As a result, a relatively small dilution factor ($D_d = 1.25$) is generally sufficient for absorbing the writes while a relatively large $D_m \times D_s$ product improves read latency. A properly configured 4-disk EW-Array halves the latency achieved on a single-disk conventional system. Note that many of the configurations in Figure 2 have fractional values for D_s and D_d, yet $D_s \times D_d$ is always integral.

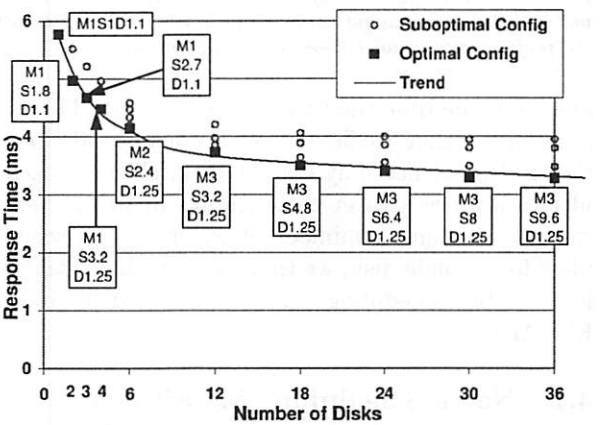


Figure 2: Comparison of response times of different EW-Array configurations. Each point symbol shows the performance of an alternative EW-Array configuration. A label "MaSbDc" denotes a $D_m \times D_s \times D_d = a \times b \times c$ configuration.

That means each replica stripes data across $D_s \times D_d$ disks. On each of those disks, only $1/D_s$ fraction of the tracks are actually used to store data, and utilization of those tracks is $1/D_d$.

Figure 3 shows how the throughput of optimally configured EW-Arrays scales with an increasing number of disks. We vary the number of outstanding requests per disk to emulate different load levels. For a fixed number of disks, as we raise the request arrival rate, a progressively larger dilution factor D_d is needed to absorb the disk writes that can no longer be masked by idle periods.

4 Scheduling on an EW-Array

When multiple outstanding requests are present in the I/O system, the order of servicing these requests has an important impact on the throughput of the system. Although disk scheduling is a well-studied problem, eager-writing presents a new challenge and a new opportunity. The challenge is that the physical locations of the write requests are un-

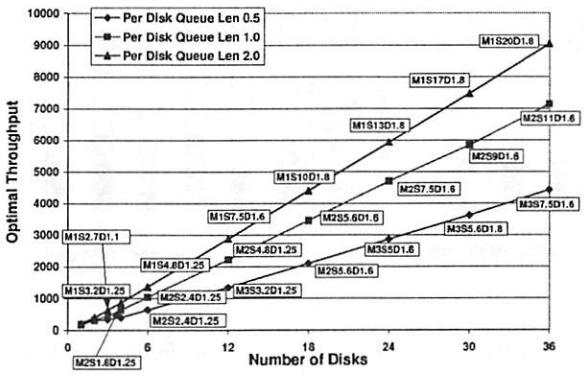


Figure 3: Throughput of optimal EW-Array configurations under different queueing conditions. Each point represents the performance of an EW-Array configuration.

known at the time the requests are queued. The opportunity is that the flexibility afforded by data location independence may enable an intelligent scheduler to achieve greater throughput. In this section, we first examine a number of eager-writing schedulers for a single disk; we then describe how we perform global scheduling across multiple disks in an EW-Array.

4.1 Naive Scheduling Algorithms

Given a mix of queued read and write requests, since write requests generally can be serviced quickly under eager-writing, one naive strategy is to simply schedule all the writes first. (To prevent starvation of read requests, one can augment this algorithm with simple heuristics such as imposing an upper-bound on the amount of time that a request can spend in the queue before it is forcibly scheduled.) We call this the *write-first* algorithm. One problem with this naive algorithm is that by greedily scheduling all the writes first, the scheduler may be missing opportunities of inserting some of these writes into naturally occurring latency gaps during the later read operations without adding much to the queueing time of the reads.

The opposite approach, an equally if not more naive algorithm, is to schedule all the reads first using an existing disk scheduling algorithm. We call this the *read-first* algorithm. Write requests that could have completed more quickly suffer long delays, and it is not hard to see why this algorithm is not optimal. We describe the read-first and write-first algorithms here not because of their practical utility, but because the problems encountered by these two extreme approaches may expose the pitfalls of eager-writing scheduling algorithms in general.

4.2 Eager-writing-based Shortest Access Time First Scheduling

The traditional shortest access time first (SATF) algorithm greedily schedules the request that is closest to the current disk head position [16, 23]. It takes both seek and rotational latency into consideration. We now extend this algorithm for eager-writing, and we call this extension the *SATF-EW* algorithm. The SATF-EW algorithm examines the queue and compares the location of the closest read request against the location of the closest free block. If the former is closer, we schedule the read request, else we schedule a write request into the closest free block. To avoid trapping the disk head in a small region and exhausting the free blocks, we always force the disk head to move in one seek direction until it can move no further and has to switch direction.

Unlike the naive algorithms described earlier, SATF-EW generally strikes a sound balance in scheduling read and write requests. When there are a large fraction of free blocks and there are many write requests, however, SATF-EW will tend to favor scheduling writes first; in the extreme case, it may degenerate to the write-first algorithm which, as we have explained earlier, may have its shortcomings.

4.3 Eager-writing-based Free Bandwidth Scheduling

"Free bandwidth" is different from bandwidth available during idle periods—the disk head may pass over locations that are of interest to some background operations even as it is "busy" serving foreground requests. Inserting some of these background requests into the foreground request stream should impose little penalty on foreground activities. Example applications that can benefit from free bandwidth are background activities such as data mining and storage reorganization [17].

Our next group of eager-writing scheduling algorithms are inspired by the approach of exploiting free bandwidth. Under this approach, we first schedule the reads using a known disk scheduling algorithm. Based on this schedule, we calculate a "deadline" by which the disk head must arrive at a target cylinder for each read operation so that the read operation can complete in time. Once the schedule and the deadlines of the reads are determined, we attempt to insert eager-writes into gaps among the reads if suitable free blocks can be located and the insertion of these eager-writes does not cause the disk head to miss the deadlines prescribed by the read schedule. In this case, the disk head also moves in one direction until it can move no further and has to reverse

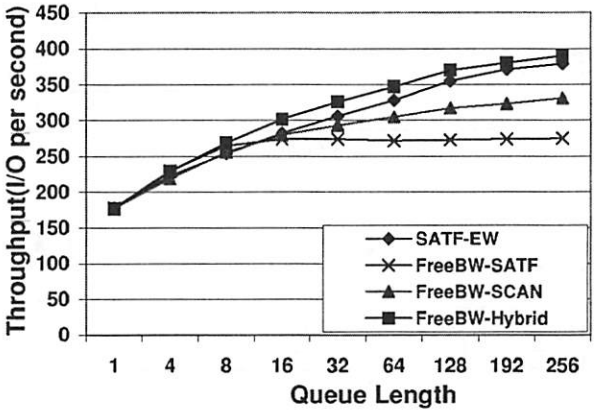


Figure 4: Throughput comparison of different eager-writing scheduling algorithms as we vary the number of queued requests.

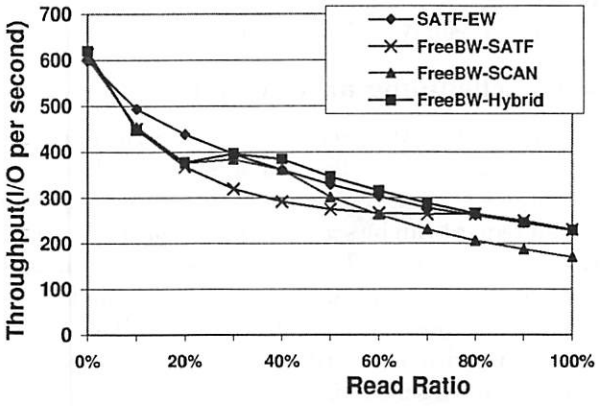


Figure 5: Throughput comparison of different eager-writing scheduling algorithms as we vary the fraction of queued operations that are reads.

its direction.

The above description is not sufficient for fully specifying the scheduling order—to complete the description, we must determine what scheduling algorithm to use to schedule the reads. We shall explore two possibilities: shortest access time first (SATF) as described above, and SCAN, which orders the read requests solely based on their seek distance. We call the resulting overall algorithms *FreeBW-SATF* and *FreeBW-SCAN* respectively.

These scheduling algorithms inspired by the exploitation of free bandwidth are a different way of balancing the scheduling of reads and eager-writes. When there are many read requests, however, these algorithms will tend to favor scheduling reads first; this happens because a large number of reads tend to reduce the gap among them and there is less room left for eager writes. In the extreme case, it may degenerate to the read-first algorithm which, as we have explained in Section 4.1, may have its shortcomings.

4.4 Comparison of Eager-writing Scheduling Algorithms

Figure 4 compares the throughput of different eager-writing algorithms as we vary the queue length. This simple simulated workload has a 50% write ratio and it runs on a disk with a dilution factor of 2.

SATF-EW works well for all queue lengths. In contrast, although it is known that SATF generally outperforms SCAN in a traditional storage system [16, 23], interestingly enough, FreeBW-SATF performs worse than FreeBW-SCAN, especially when the queue is large. This occurs because the aggressive scheduling of reads by SATF under high load leaves little "free bandwidth" for scheduling the eager-writes; consequently, FreeBW-SATF becomes similar to the read-first algorithm and unnecessarily penalizes writes. In contrast, by using SCAN to schedule the reads, FreeBW-SCAN causes the reads to be spaced further apart so more "free bandwidth" becomes available for eager-writes; consequently, the scheduling of reads and writes are better balanced and the overall performance of FreeBW-SCAN is better.

When there are a large number of reads but few writes, however, the performance of FreeBW-SCAN is not the best due to its failure to take rotational delay of reads into consideration. To address this shortcoming, we augment FreeBW-SCAN with a simple heuristics: when there is no write request in the queue, we replace SCAN with SATF. We call the modified algorithm *FreeBW-Hybrid*. Figure 4 shows that this hybrid algorithm performs the best for this workload—it even slightly outperforms SATF-EW due to its successful masking the eager-writes in the gaps of reads.

Figure 5 compares the throughput of these algorithms as we vary the ratio of reads. The queue length is 64 and the disk dilution factor is 2.

SATF-EW works well for all read ratios. When the read ratio is high, the disadvantage of FreeBW-SCAN is most apparent. In contrast, all other algorithms approach the performance of SATF when the read ratio approaches 100%. Interestingly, when the number of reads is small (but nonzero), the free bandwidth-based approaches also perform worse than SATF-EW. This is because the seek time among those small number of reads dominates and there is little rotational time left for scheduling the eager-writes. When there are a modest number of reads, due to both of its ability of successfully ex-

ploiting free bandwidth and its intelligent scheduling of reads, FreeBW-Hybrid is the best.

4.5 Scheduling an EW-Array

So far, we have described how to perform eager-writing-based scheduling on a single disk. The scheduling on an EW-Array is more complex because a read request can be serviced by any one of the disks that has a replica and a write request can return as soon as the first one or two copies are made persistent. We now incorporate the single-disk eager-writing algorithms described in the previous sections into the mirror scheduling algorithm employed by Yu et al [30].

A read request is sent to the idle disk head that is closest to a replica if at least one of the disks that contain the data is idle. If all the disks that contain the desired data are busy, a duplicate request is inserted into each of the relevant drive queues. Each disk employs an eager-writing scheduling algorithm as described in the previous sections. As soon as one of the duplicates completes, all remaining duplicate requests are canceled.

A write request can be sent to any one of the disks that are supposed to contain a replica. If any of these disks are idle, it is sent to the one that is closest to a free block. If all disks that should contain the desired data are busy, the request is inserted into the shortest queue. A second foreground write can be similarly scheduled for increased reliability. We set aside the remaining replica writes (if any) in a separate *delayed write queue* associated with each drive. Replica propagations from the delayed write queues are scheduled only when the foreground queues are empty.

5 Implementation

In this section, we describe a prototype EW-Array implementation that we use to experiment with the configuration and scheduling alternatives.

5.1 Architecture

The EW-Array prototype is implemented on the "MimdRAID" system developed by Yu et al [30]. Figure 6 shows how some of the MimdRAID modules have been replaced by EW-Array-specific components and how these components fit together. MimdRAID provides a framework and a number of useful features that enable one to conveniently experiment with novel disk array designs. We briefly highlight some of these useful features:

- MimdRAID exports a transparent logical disk interface on Windows 2000 so that the existing operating system and applications run unmodified on the underlying experimental disk array.
- Highly modularized components of MimdRAID, such as the "Array Configuration" and "Scheduling" components of Figure 6, can be replaced to allow for experimentation with new array designs.
- An accurate software-based disk head position prediction mechanism (in the "Calibration" layer) is crucial for realizing an EW-Array because the efficient scheduling of both eager-writes and reads depends on the precise knowledge of the rotational position of the disk head.
- At the lowest layer, the "SCSI Abstraction" module, which manages real Seagate disks, can be substituted with a disk simulator, so an EW-Array simulator and its implementation effectively share most of the code. The simulator can shorten simulation time of long traces by replacing physical I/O time with simulated time; it also allows us to experiment with a wide range of configurations without having to physically construct them.

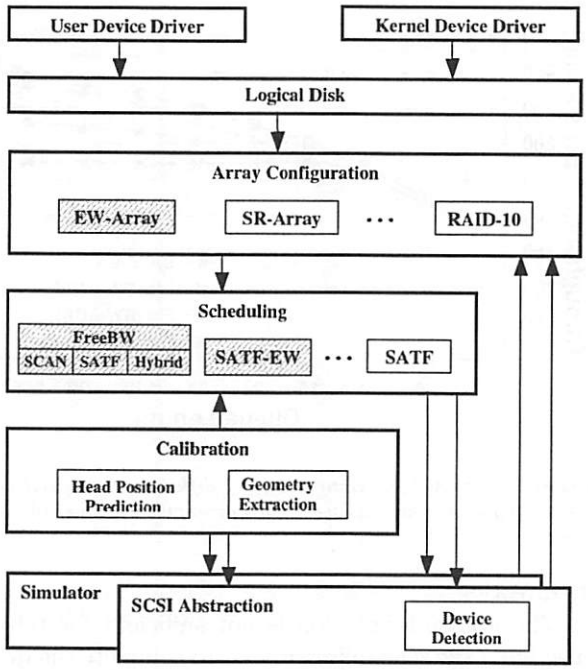


Figure 6: The MimdRAID architecture. Shaded parts are the modules added for an EW-Array.

Almost all the EW-Array-specific code in MimdRAID is concentrated in the Scheduling and Array Configuration layers. The Scheduling layer implements all the eager-writing scheduling algorithms described in Section 4. The Array Configuration layer is responsible for translating requests of logical I/O addresses to those of physical I/O ad-

dresses. This layer maintains a logical-to-physical address mapping and we describe this next.

5.2 Logical-to-physical Address Mapping

Under eager writing, the physical locations of a logically addressed block can frequently change. In this section, we detail how we query the logical-to-physical address mapping upon a read operation, how we maintain persistence of the mapping upon a write operation, and how we recover the mapping.

5.2.1 Querying the Mapping

A simple design is to store the entire logical-to-physical mapping in main memory. Reads are efficient and simple to implement: for each read operation, we simply use the logical address as an index to query a table to uncover the physical addresses. The price we pay is the cost of the map memory. A map entry in our system consumes four bytes per logical address per replica. The block size of our EW-Array implementation is 4 KB. With a $D_d \times D_m \times D_s$ EW-Array, the amount of map space is $D_m \cdot C/1000$, where C is the size of the logical disk which, in turn, is typically much smaller than the total amount of physical capacity in a TPC-C run.

We have chosen this simple design due to the nature of the transaction processing workload that we are targeting. First, the large number of spindles that are necessary for achieving good performance make the cost of the map memory insignificant. Second, the poor locality of the workload implies that the relatively small amount of memory consumed by the map would have delivered little improvement to read performance had the memory been used as a data cache instead. For a workload that exhibits more locality, we are currently researching the alternative approach of keeping only the most frequently accessed portion of the map in memory and "paging" the rest to disk.

5.2.2 Updating and Recovering the Mapping with Incremental Checkpointing

In designing a mechanism to keep the logical-to-physical address map persistent, we strive to accomplish two goals: one is low overhead imposed by the mechanism on "normal" I/O operations, and the other is fast recovery of the map.

Figure 7 shows the map-related data structures used in various storage levels. Updates to the logical-to-physical map are accumulated in a small amount of NVRAM. When the NVRAM is filled, its content is appended to a *map log region* on disk. (Both

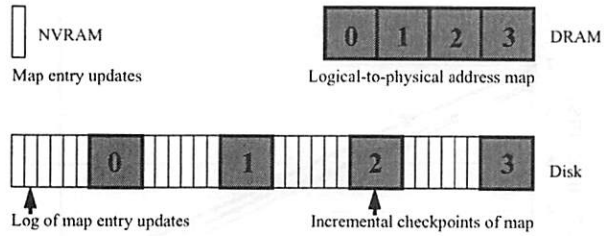


Figure 7: Logging of logical-to-physical map updates and incremental checkpointing of the map.

the NVRAM and the map log region on disk can be replicated for increased reliability.) We divide the logical-to-physical address map into M portions. ($M = 4$ in Figure 7.) Periodically, a portion of the map is checkpointed as it is appended to the log. After all M portions of the map are checkpointed, the map update log entries that are older than the Mth oldest checkpoint can be freed and this freed space can be used to log the new updates. The size of the map log region on disk is bound by the frequency and the size of the checkpoints. When we reach the end of the log region, the log can simply "wrap around" since we have the knowledge that the log entries at the beginning of the region must have been freed. The location of the youngest valid checkpoint is also stored in NVRAM.

During recovery, we first read the entire map log region on disk to reconstruct the in-memory logical-to-physical address map, starting with the youngest valid checkpoint. We then replay the log entries buffered in NVRAM. At the end of this process, the in-memory logical-to-physical address map is fully restored.

We note a number of desirable properties of the mechanism described above. The size and frequency of the checkpoint allows one to trade off the overhead during normal operation against the map recovery time. In particular, the checkpoints bound the size of the map log region, thus bounding the recovery time. Incremental checkpointing prevents undesirable prolonged delays associated with the checkpointing of the entire map. It also allows the space occupied by obsolete map update entries to be reclaimed without expensive garbage collection of the log.

It is interesting to compare the mechanism employed in managing the logical-to-physical address map of an EW-Array against those employed in managing data itself in a number of related systems such as an NVRAM-backed LFS [1, 21] and RAPID-Cache [13]. While we buffer and checkpoint *metadata*, these other systems buffer and reorganize *data*. The three orders of magnitude difference in

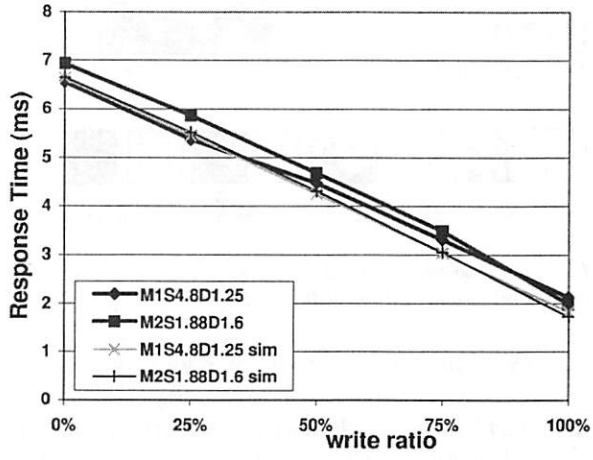


Figure 8: Comparison of response times on the EW-Array prototype and those predicted by the simulator as we vary the read/write ratio.

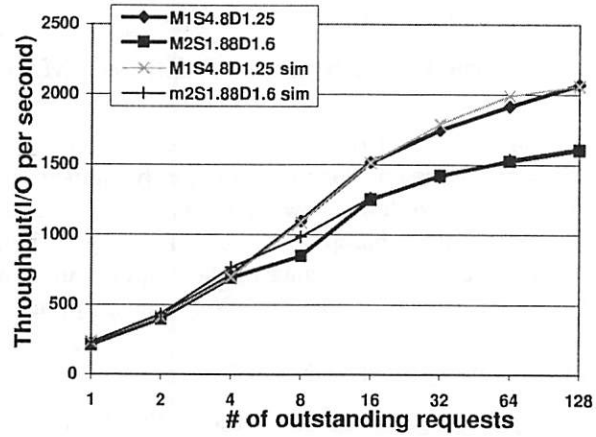


Figure 9: Comparison of throughput on the EW-Array prototype and that predicted by the simulator as we vary the per-disk queue length.

the number of bytes involved makes it easy to control the overhead of managing metadata in an EW-Array. Practically, in a $D_m \times D_s \times D_d = 2 \times 9 \times 2$ EW-Array prototype (whose disks are 9 GB each), with a 100KB NVRAM to buffer map entry updates, we have observed that the amount of overhead due to the maintenance of the map during normal operations of the TPC-C workload is below 1% and it takes 9.7 seconds to recover the map. The recovery performance can be further improved if we distribute the map log region across a number of disks so the map can be read in parallel.

5.3 Validating the Integrated Simulator

Operating system	Microsoft Windows 2000
CPU type	Intel Pentium III 733 MHz
Memory	128 MB
SCSI Interface	Adaptec 39160
SCSI bus speed	160 MB/s
Disk model	Seagate ST39133LWV 9.1 GB
RPM	10000
Average seek	5.2 ms read, 6.0 ms write

Table 1: Platform characteristics.

Due to the large number of configurations and the long traces that we must experiment with, the experimental results reported in Section 6 are based on those obtained on the simulator; therefore it is necessary to validate the EW-Array simulator using our EW-Array prototype. Table 1 lists some of the platform characteristics of the prototype.

We run a benchmark called "Iometer", a benchmark developed by the Intel Server Architecture Lab [15]. Iometer can generate workloads of various characteristics including read/write ratio, request size, and the maximum number of outstanding requests.

Figure 8 compares the response times measured on a number of six-disk EW-Array prototype configurations with those predicted by the simulator as we vary the read/write ratio. In these Iometer experiments, the number of outstanding requests is one, and the dilution factor of the EW-Array is two. The two EW-Array configurations ($D_m \times D_s \times D_d = 1 \times 3 \times 2$, and $D_m \times D_s \times D_d = 2 \times 1.5 \times 2$) have similar response times and they are closely matched by those predicted by the simulations. Since the eager-writes have much lower latency than reads, the response time decreases as the write ratio increases.

Figure 9 compares the throughput obtained on the same six-disk EW-Array configurations with that predicted by the simulator as we vary the queue length per disk. In these Iometer experiments, the write ratio is 50%. As the per-disk queue length increases, the $1 \times 4.8 \times 1.25$ EW-Array achieves greater throughput than the $2 \times 1.8 \times 1.6$ configuration because it becomes increasingly difficult for the latter configuration to mask the replica propagation even with a larger dilution factor. The throughput measured on the prototype matches closely the simulated result.

6 Experimental Results

In this section, we compare the performance of the EW-Array with that of a number of alternatives.

6.1 The TPC-C Trace

The eager-writing arrays are designed to target TPC-C-like transaction applications. We evaluate

the effectiveness of EW-Arrays with a trace supplied by HP Labs. It is a disk trace of an unaudited run of the Client/Server TPC-C benchmark running at approximately 1150 tpmC on a 100-warehouse database. It has 4.2 million I/O requests, 54% of which are reads. The I/O rate is about 700 I/Os per second in the steady state. Most of the requests are synchronous I/Os. The total data set is about 9 GB, distributed originally on 54 disks to achieve the desired throughput. The trace was collected on 5/03/94.

Two characteristics of the trace may be of concern due to its old age: the data rate and the size of the data set. With comparable number of disks and machines, the current technology can support a much higher data rate. To account for this development, in some of the following experiments, we raise the I/O rate by multiplying it with a "trace speedup" factor. For example, when the trace speedup is two, we halve the inter-arrival time of requests. The data set size factor is of less concern. In fact, only a small fraction of the space on the original traced disks was used to achieve the target I/O rate. Although a single modern disk can accommodate the entire traced data set today, it cannot support the data rate of the original trace. We shall vary the number of disks employed in a disk array onto which the traced data set is distributed. We study the effectiveness of various array configurations and the conclusions that we reach are independent of the size of the entire data set.

6.2 The Alternative Disk Array Configurations

In addition to the EW-Array configurations, we will experiment with the following alternatives:

- A *RAID-10* combines striping and mirroring: data is striped across a number of disks and each of the striped disks is also replicated once.
- A *Doubly Distorted Mirror* is a variant of a RAID-10. For each logical write request, two "write-anywhere" physical writes are performed to free locations near the disk heads. One of these two copies is later "moved" to a fixed location in the background [19].
- An *SR-Array* combines striping and rotational replication: data is striped across D_s disks, and each block is replicated D_r times within a track to reduce rotational delay, so a total of $D_r \times D_s$ disks are used [30].

For configurations that support replication on multiple disks, we shall experiment with two different reliability guarantee scenarios: in one scenario, a synchronous write request is allowed to return as soon as one physical write is committed to a disk platter; in a second scenario, a synchronous write request is not allowed to return until at least two physical writes are committed to two disks. When multiple requests are in the disk queue, the EW-Arrays employ the SATF-EW scheduler discussed in Section 4, and the other configurations employ variants of SATF.

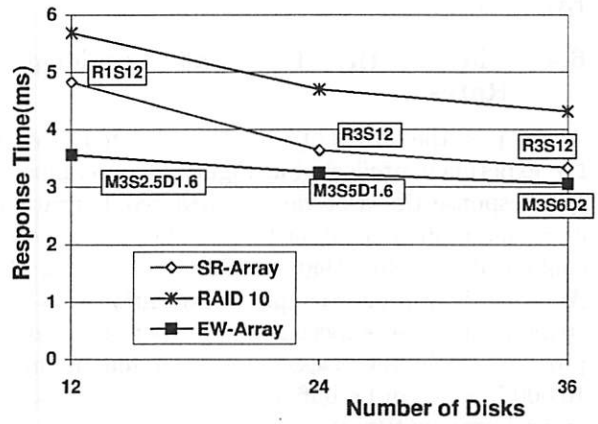


Figure 10: Comparison of TPC-C I/O response time on several disk array configuration alternatives as we vary the number of disks in the array. The SR-Arrays and EW-Arrays are labeled with their configuration parameters: an "RaSb" label denotes a $D_s \times D_r = b \times a$ SR-Array configuration, and an "MaSbDc" label denotes a $D_m \times D_s \times D_d = a \times b \times c$ EW-Array configuration.

6.3 Playing the Trace at Original Speed

We play the TPC-C trace at original speed in the experiments reflected in Figure 10. It compares I/O response times of the optimally configured EW-Arrays against those of the RAID-10s and the optimally configured SR-Arrays as we increase the number of disks. In these experiments, the second and subsequent replicas, if any, are propagated in the background.

An SR-Array generally outperforms RAID-10 because its combination of striping and rotational replication balances the reduction of seek and rotational delays better. An EW-Array outperforms both because of its substantially lower write latency, which also enables a higher degree of replication, which in turn lowers read latency. As we increase the number of disks, the performance benefit derived by an SR-Array from an increasing number of disks is larger than that of an EW-Array because the aggressive rotational delay reduction of the former benefits both reads and writes, while the write latency on an EW-Array is small to begin with and further improvements are marginal. In general, far fewer disks are necessary to achieve a specific latency goal on an

EW-Array.

6.4 Playing the Trace at Accelerated Rates

We play the TPC-C trace at accelerated rates in the experiments reflected in Figure 11. It compares I/O response times of various array configurations as we maintain a constant total of 36 disks for each configuration. (Note that the $D_s \times D_r = 36 \times 1$ SR-Array configuration is simply a conventional 36-way stripe.) In these experiments, we perform replica propagation in the background. (A maximum of 10,000 blocks can be buffered on an array for background propagation.)

The EW-Array has the best response time under all arrival rates and it generally delivers much higher sustainable throughput than conventional configurations. For example, the maximum sustainable throughput rates (expressed in terms of the speedup rate over the original trace speed) on a 36-way stripe, a RAID-10, a $2 \times 9 \times 2$ EW-Array, and a $1 \times 22 \times 1.6$ EW-Array are approximately 5×, 8×, 10×, and 14×, respectively.

As we raise the request arrival rate, idle time becomes more scarce and the replica propagation cost becomes felt by all configurations. We must successively reduce the degree of replication for both the SR-Array and the EW-Array. Thanks to the very low write latency of the EW-Array, however, the replica propagation burden on the EW-Array represents a much lighter load. The $2 \times 9 \times 2$ EW-Array remains a configuration of choice that offers sub-5 ms response times for an arrival rate that is as high as 9× the original, a rate that has rendered replica propagation a costly luxury that the other approaches cannot afford. The payoff of replication is better read response time than that on the other configurations.

In addition to eager-writing, two other factors contribute to the EW-Array's superior throughput. One is the greater flexibility afforded by its SATF-EW local disk scheduler. The other is the better load-balancing opportunities afforded by the array-wide scheduling heuristics as writes are scheduled to disks with shorter queues and reads are serviced by choosing among multiple candidate copies on different disks.

6.5 Effect of Double Synchronous Writes

In the previous experiments, all but the first replicas are propagated in the background. To raise the degree of reliability, one may desire to have two copies physically on disk before a write request is

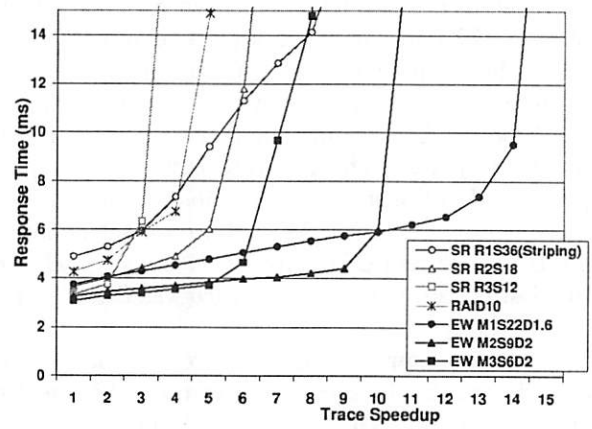


Figure 11: Comparison of TPC-C throughput on alternative disk array configurations as we vary the I/O rate. The total number of disks is a constant (36). The size of the delayed write buffer is 10,000 blocks.

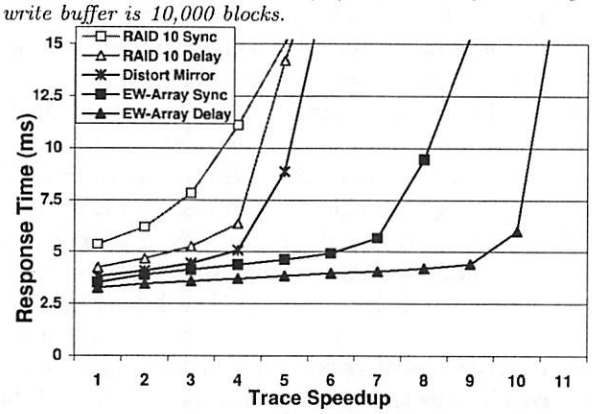


Figure 12: Effect on TPC-C throughput as we write to two disks synchronously. The total number of disks is a constant (36). Labels that include the word "Delay" denote experiments that propagate all but the first replicas in the background. Labels that include the word "Sync" denote experiments that write two replicas synchronously.

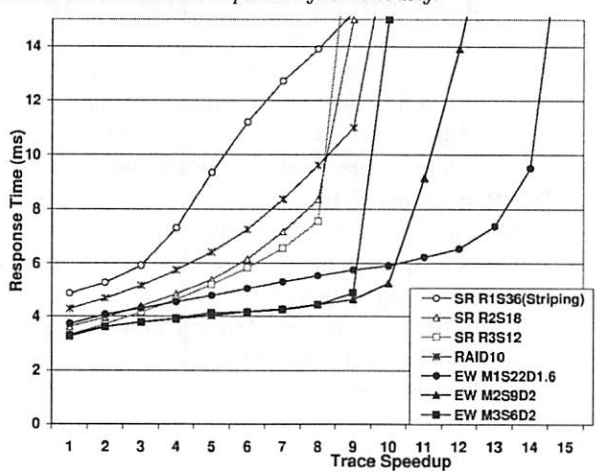


Figure 13: Effect on TPC-C throughput as we increase the size of the delayed write buffer to 100,000 blocks. The total number of disks is a constant (36).

allowed to return. We now study the effect of increasing foreground writes on three alternative configurations: a RAID-10, a Doubly Distorted Mirror (DDM), and a $2 \times 9 \times 2$ EW-Array. The DDM performs two synchronous eager-writes and moves one of the two copies to a fixed location in the background. We note that the system that we have called the DDM is in fact a highly optimized implementation that is based on the MimdRAID disk location prediction mechanism and the eager-writing scheduling algorithms, features not detailed in the words of "write anywhere" in the simple original simulation-based study [19]. (We do not consider SR-Arrays because the pure form of an SR-Array involves only intra-disk replication which does not increase the reliability of the system.)

As expected, extra foreground write slows down both the RAID-10 and the EW-Array. However, for a given request arrival rate that does not cause performance collapse, the response time degradation experienced by the RAID-10 is more pronounced than that seen on the EW-Array. This is because the cost of an extra update-in-place foreground write is relatively greater than that of an extra foreground eager-write. The performance of the DDM lies in between , because the two foreground writes enjoy some performance benefit of eager-writing but the extra update-in-place write becomes costly, especially when the request arrival rate is high. One of the purposes of this third update-in-place write is to restore data locality that might have been destroyed by the eager-writes. This is useful for workloads that exhibit both greater burstiness and locality. Unfortunately, the TPC-C workload is such that it does not benefit from this data reorganization.

6.6 Effect of the Delayed Write Buffer Size

We have seen that replica propagation imposes a significant cost on update-in-place-based disk arrays such as RAID-10 and SR-Array. One possible way of alleviating this burden to make these alternatives more attractive is to use a larger delayed write buffer. A larger delayed write buffer is useful in two ways. One is that it may allow larger batches of replica propagations to be scheduled and these larger batches can utilize the disk bandwidth more efficiently. The second source of efficiency is that a larger buffer can potentially more effectively smooth the burstiness so that replica propagation does not have to occur in the foreground due to lack of buffer space.

Figure 13 shows the results of repeating the throughput experiments shown in Figure 11 after

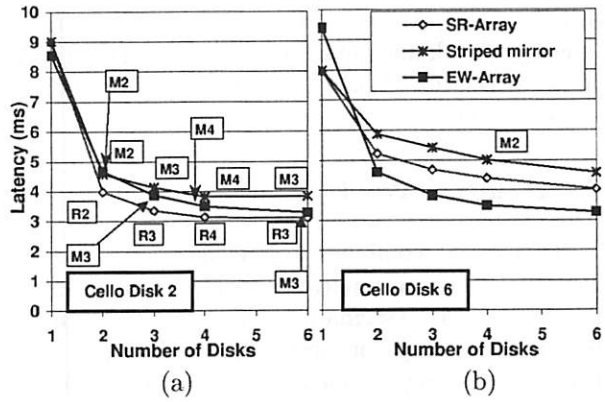


Figure 14: Comparison of the performance of different array alternatives under two file system workloads: (a) Cello disk 2, housing "/users", and (b) Cello disk 6, housing "/var/spool/news". Each data point represents the performance of the best configuration based on a given array alternative. The rectangular labels show the degree of mirroring (or replication) used in the configurations. The unlabeled configurations in the second figure have identical degrees of mirroring (or replication) as their counterparts in the first figure.

we have increased the delayed write buffer from 10,000 blocks to 100,000 blocks. As expected, the curves representing array configurations that require data replication have all shifted to the right, signaling higher maximum sustainable throughput rates. However, even with this aggressive delayed write buffering, the high cost of update-in-place is still apparent and the advantage of eager-writing is still significant.

6.7 Results of File System Workloads

Although the target workload of an EW-Array is TPCC-like transaction processing applications, it is natural to ask whether it works for other workloads. Figure 14 shows the performance results of two file system workloads that are selected from the HP "Cello" trace. Cello is a two month trace of a file server supporting simulations, compilations, editing, and reading mail and news. We use the traces of two disks during the week from 5/30/92 to 6/6/92. Disk 2 houses user home directories and disk 6 houses a news archive. To compensate for the relatively lower I/O rates of the trace, we speed up the trace playing rate by a factor of four.

The difference between Figure 14(a) and (b) is due to the different locality and the different read/write ratio of the two workloads. Cello disk 2 exhibits a higher degree of locality: the average seek distance on this disk is about half of that of disk 6. Cello disk 6, on the other hand, experiences a higher write ratio: 63.2% on disk 6 versus 45.2% on disk 2. Therefore, an SR-Array performs best for

disk 2 by preserving locality and balancing the reduction of seek and rotational delays; while an EW-Array excels for disk 6 by aggressively optimizing write latency.

7 Related Work

This paper combines four elements: (1) eager-writing, (2) data replication for performance improvement, (3) systematic configuring a disk array to trade capacity for performance, and (4) intelligent scheduling of queued disk requests. While each of these techniques can individually improve I/O performance, it is the integration and interaction of these techniques that allow one to economically achieve scalable performance gain on TPC-C-like transaction processing workloads. We briefly describe some related work in each of those areas.

The eager-writing technique dates back to the IBM IMS Write Ahead Data Set (WADS) system which writes write-ahead log entries in an eager-writing fashion on drums [7]. Hagmann employs eager-writing to improve logging performance on disks [10]. A similar technique is used in the Trail system [14]. These systems require the log entries to be rewritten to fixed locations. Mime [4], the extension of Loge [6], integrates eager-writing into the disk controller and it is not necessary to rewrite the data created by eager-writing. The Virtual Logging Disk and the Virtual Logging File Systems eliminate the reliance on NVRAM for maintaining the logical-to-physical address mapping and further explore the relationship between eager-writing and log-structured file systems [28]. All of these systems work on individual disks.

A more common approach to improving small write performance is to buffer data in NVRAM and periodically flush the full buffer to disk. The NVRAM data buffer provides two benefits: efficient scheduling of the buffered writes, and potential overwriting of data in the buffer before it reaches disk. For many transaction processing applications, poor locality tends to result in few overwrites in the buffer, and lack of idle time makes it difficult to mask the time consumed by buffer flushing. It is also difficult to build a large, reliable and inexpensive NVRAM data buffer. On the other hand, an inexpensive reliable small NVRAM buffer, as the one employed for the mapping information in the EW-Array, is quite feasible.

The systems that use the NVRAM data buffer differ in the way they flush the buffer to disk. The conventional approach is to flush data to an update-in-place disk. In the steady state, the throughput of such a system is limited by the average head movement distance between consecutive disk writes. An alternative to flushing to an update-in-place disk is to flush the buffered data in a log-structured manner [1, 21]. The disadvantage of such a system is its high cost of disk garbage collection, which is again exacerbated by the poor locality and the lack of idle time in transaction processing workloads [22, 24]. Some systems combine the use of a log-structured "caching disk" with an update-in-place data disk [12, 13], and they are not immune to the problems associated with each of these techniques, especially when faced with I/O access patterns such as those seen in TPC-C.

Several systems are designed to address the small write performance problem in disk arrays. The Doubly Distorted Mirror (DDM) [19] is closest in spirit to the EW-Array. The two studies have different emphasis, though. First, the emphasis of the EW-Array study is to explore how to balance the excess capacity devoted to eager-writing, mirroring, and striping, and how to perform disk request scheduling in the presence of eager-writes. Second, the EW-Array employs pure eager-writing without the background movement of data to fixed locations. While this is more suitable for TPC-C-like workloads, other applications may benefit from a data reorganizer. Third, the EW-Array study provides a real implementation.

While the DDM and the EW-Array are based on mirrored organizations, the techniques that may speed up small writes on individual disks may be applied to parity updates in a RAID-5. *Floating parity* employs eager-writing to speed up parity updates [18]. *Parity Logging* employs an NVRAM and a logging disk to accumulate a large amount of parity updates that can be used to recompute the parity using more efficient large I/Os [25]. The amount of performance improvement experienced by read requests in a RAID-5 is similar to that on a striped system, and as we have seen in the experimental results, a striped system may not be as effective as a mirrored system, particularly if the replica propagation cost of a mirrored system is reduced by eager writing.

Instead of forcing one to choose between a RAID-10 and a RAID-5, the AutoRAID combines both so that the former acts as a cache of the latter [29]. The RAID-5 lower level is log-structured and it employs a *hole-plugging* technique for efficiently garbage-collecting a nearly full disk: live data is "plugged" into free space of other segments. This is similar to eager-writing, except that eager-writing does not require garbage collection.

An SR-Array combines striping with rotational

data replication to reduce both seek and rotational delay [30]. A mirrored system may enjoy some similar benefits [2, 5]. Both approaches allow one to trade capacity for better performance. The difficulty in both cases is the replica propagation cost. The relative sizing of the two levels of storage in AutoRAID is a different way of trading capacity for performance. In fact, for locality-rich workloads, it is possible to employ an SR-Array or an EW-Array as an upper-level disk cache of an AutoRAID-like system.

Seltzer and Jacobson have independently examined disk scheduling algorithms that take rotational delay into consideration [16, 23]. Yu et al. have extended these algorithms to account for rotational replicas [30]. Polyzois et al. have proposed a delayed write scheduling technique for a mirrored system to maximize throughput [20]. Lumb et al. have exploited the use of "free bandwidth" for background I/O activities [17]. The EW-Array scheduling algorithms have incorporated elements of these previous approaches.

Finally, the goal of the MimdRAID project is to study how to configure a disk array system given certain cost/performance specifications. The "attribute-managed storage" project at HP [8] shares this goal.

8 Conclusion

Due to their poor locality, high update rates, lack of idle time, and high reliability requirements, transaction processing application such as those exemplified by TPC-C are among the most demanding I/O applications. In this paper, we have explored how to integrate eager-writing, mirroring, and striping in a eager-writing disk array design that effectively caters to the need of these applications. Mirroring and striping improves read performance, while eager-writing improves write performance and reduces the cost of data replication. The combination provides a high degree of reliability without imposing excessive performance penalty. To fully realize the potential of an EW-Array, we must address two issues. One is the careful balance of extra disk capacity that is devoted to each of the three dimensions: free space dilution for eager-writing, the degree of mirroring, and the degree of striping. The second is the intelligent scheduling of the queued requests so that the flexibility afforded by the high degree of location independence associated with eager-writing is fully exploited. Simulation and implementation results indicate that the prototype EW-Array can deliver latency and throughput results unmatched by conventional approaches for an important class of transaction processing applications.

Acknowledgement

We thank Ruoming Pang for proposing the incremental metadata checkpointing algorithm, Fengzhou Zheng for tirelessly running many of the experiments, Sumeet Sobti for editing an earlier draft, Walter Burkhard for careful shepherding of the paper, and the FAST PC members and reviewers for their excellent suggestions.

References

[1] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-Volatile Memory for Fast, Reliable File Systems. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (Sept. 1992), pp. 10–22.

[2] Bitton, D., and Gray, J. Disk Shadowing. In *Proc. of the Fourteenth International Conference on Very Large Data Bases* (Los Angeles, CA, August 1988), Morgan Kaufmann, pp. 331–338.

[3] Borr, A. Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing. In *Proc. of the Seventh International Conference on Very Large Data Bases* (Cannes, France, September 1981), IEEE Press, pp. 155–165.

[4] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA, March 1992.

[5] Dishon, Y., and Lui, T. S. Disk Dual Copy Methods and Their Performance. In *Proc. of Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)* (Tokyo, Japan, 1988), IEEE CS Press, pp. 314–318.

[6] English, R. M., and Stepanov, A. A. Loge: a Self-Organizing Disk Controller. In *Proc. of the 1992 Winter USENIX* (January 1992).

[7] Gawlick, D., Gray, J., Limura, W., and Obermarck, R. Method and Apparatus for Logging Journal Data Using a Log Write Ahead Data Set. U.S. Patent 4507751 issued to IBM, March 1985.

[8] Golding, R., Shriver, E., Sullivan, T., and Wilkes, J. Attribute-managed Storage. In *Workshop on Modeling and Specification of I/O* (San Antonio, TX, October 1995).

[9] Growchowski, E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In *Datatech* (September 1998), ICG Publishing, pp. 11–16.

[10] HAGMANN, R., AND GARCIA-MOLINA, H. Implementing Long Lived Transactions Using Log Record Forwarding. Tech. Rep. CSL-91-2, Xerox Corporation, Palo Alto, CA, February 1991.

[11] HSIAO, H.-I., AND DEWITT, D. J. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 1990 IEEE International Conference on Data Engineering* (February 1990), pp. 456–465.

[12] HU, Y., AND YANG, Q. DCD—Disk Caching Disk: A New Approach for Boosting I/O Performance. In *Proc. of the 23rd International Symposium on Computer Architecture* (1995), pp. 169–178.

[13] HU, Y., YANG, Q., AND NIGHTINGALE, T. RAPID-Cache—A Reliable and Inexpensive Write Cache for Disk I/O Systems. In *Proc. of the Fifth International Symposium on High Performance Computer Architecture (HPCA-5)* (Orlando, Florida, January 1999).

[14] HUANG, L., AND CHIUEH, T. Trail: Write Optimized Disk Storage System. http://-www.ecsl.cs.sunysb.edu/trail.html.

[15] INTEL SERVER ARCHITECTURE LAB. Iometer: The I/O Performance Analysis Tool for Servers. http://developer.intel.com/design/servers-/devtools/iometer.

[16] JACOBSON, D. M., AND WILKES, J. Disk Scheduling Algorithms Based on Rotational Position. Tech. Rep. HPL-CSP-91-7rev1, Hewlett-Packard Company, Palo Alto, CA, February 1991.

[17] LUMB, C., SCHINDLER, J., GANGER, G. R., RIEDEL, E., AND NAGLE, D. F. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[18] MENON, J., ROCHE, J., AND KASSON, J. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing 17*, 1 and 2 (January/February 1993), 129–139.

[19] ORJI, C. U., AND SOLWORTH, J. A. Doubly Distorted Mirrors. In *Proc. of ACM SIGMOD Conference* (May 1993), pp. 307–316.

[20] POLYZOIS, C., BHIDE, A., AND DIAS, D. Disk Mirroring with Alternating Deferred Updates. In *Proc. of the Nineteenth International Conference on Very Large Data Bases* (Dublin, Ireland, 1993), Morgan Kaufmann, pp. 604–617.

[21] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. In *Proc. of the 13th Symposium on Operating Systems Principles* (Oct. 1991), pp. 1–15.

[22] SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[23] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk Scheduling Revisited. In *Proc. of the 1990 Winter USENIX* (Washington, D.C., Jan. 1990), Usenix Association, pp. 313–323.

[24] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[25] STODOLSKY, D., HOLLAND, M., AND GIBSON, G. A. Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proc. of the 21st International Symposium on Computer Architecture* (1993), pp. 64–75.

[26] TERADATA CORP. *DBC/1012 Database Computer System Manual Release 2.0*, November 1985.

[27] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC Benchmark C Standard Specification*. Waterside Associates, Fremont, CA, August 1996.

[28] WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Virtual Log Based File Systems for a Programmable Disk. In *Proc. of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), Operating Systems Review, Special Issue, pp. 29–43.

[29] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems 14*, 1 (February 1996).

[30] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading Capacity for Performance in a Disk Array. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association

Addison-Wesley
Kit Cosper
Earthlink Network
Edgix
Interhack Corporation
Interliant
Lessing & Partner
Linux Security, Inc.
Lucent Technologies
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
Nimrod AS
O'Reilly & Associates Inc.
Raytheon Company
Sams Publishing
The SANS Institute
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
TechTarget.com
UUNET Technologies, Inc.

Supporting Members of SAGE

Certainty Solutions
Collective Technologies
Electric Lightwave, Inc.
ESM Services, Inc.
Lessing & Partner
Linux Security, Inc.
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Publishing
O'Reilly & Associates Inc.
Raytheon Company
Remedy Corporation
RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-880446-03-0